Functional Query Languages with Categorical Types

Functional Query Languages with Categorical Types

A dissertation presented

by

Ryan Wisnesky

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosphy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

November 2013

Dissertation Advisor: Professor Greg Morrisett Ryan Wisnesky

Functional Query Languages with Categorical Types

Abstract

We study three category-theoretic types in the context of functional query languages (typed λ-calculi extended with additional operations for bulk data processing). The types we study are:

- *The dependent identity type.* By adding identity types to the simply-typed λ-calculus we obtain a language where embedded dependencies are first-class objects that can be manipulated by the programmer and used for optimization. We prove that the chase re-writing procedure is sound for this language.
- *The type of propositions.* By adding propositions to the simply-typed λ-calculus, we obtain higher-order logic. We prove that every hereditarily domain-independent higher-order logic program can be translated into the nested relational algebra, thereby allowing higher-order logic to be used as a query language and giving a higher-order generalization of Codd's theorem.
- *The type of finitely presented categories.* By adding types for finitely presented categories to the simply-typed λ-calculus we obtain a schema mapping language for the functorial data model. We define FQL, the first query language for this data model, investigate its metatheory, and build a SQL compiler for FQL.

Contents

Chapter 1

Introduction

Our thesis concerns *embedded query languages* [79]: λ-calculi enriched with additional operations for bulk data processing. Such languages are typically used to query large databases and are distinguished from more general programming languages in three ways:

- Because databases may be large, embedded query languages typically expose only those bulk operations that can be implemented efficiently.
- Because databases may be physically stored in many different ways, embedded query languages are typically compiled by rapidly searching the space of equivalent queries as guided by a detailed physical cost model.
- Because the purpose of the bulk operations is to query a database, not to update it, embedded query languages are typically purely functional and embedded in a more powerful, potentially impure update language.

A traditional example of an embedded query language is SQL, and SQL queries are often embedded in the more powerful PSM (persistent stored modules [37]) update language. SQL is compiled by translation into a physical operator algebra as guided by detailed statistics about underlying data, and until the SQL-1999 standard it omitted expensive operations such as transitive closure [37]. More recently, the emergence of so-called "no-SQL" systems [77] has led to a proliferation of non-relational embedded query languages such as MapReduce [26] and embedding languages such as Pig [67] and Jaql [12].

Our thesis concerns a particular class of embedded query languages: the *functional query languages* [43]. These languages further specialize embedded query languages in two ways:

- The syntax of these languages typically derives from some form of type theory [64].
- The semantics of these languages typically derives from some form of category theory [9].

The first functional query language was the nested relational calculus (NRC) [82]. The NRC extends the simply-typed lambda calculus with a type of finite sets and operations for manipulating sets. The particular set-theoretic operations of the NRC—empty set, singleton, and union—were chosen because they correspond to the categorical notion of a monad [54]. Since then, many functional query languages have been proposed, including the monad comprehension calculus [41], Data Parallel Haskell [19], and Links [23].

The purpose of our thesis is to investigate several types that have never before been applied to functional query languages. For each type, we show why that type is useful for information management, prove foundational theorems about the type, and build software illustrating how the type may be used in practice. The three types we investigate are:

- *The dependent identity type.* In chapter 2, we add identity types [11] to the nested relational calculus and obtain a functional query language where data integrity constraints are first-class objects that can be manipulated by the programmer and used for query optimization. We prove that embedded dependencies [2] can be represented as identity types, and that the well-known chase optimization procedure [70] is sound for this language.
- *The type of propositions.* In chapter 3, we add propositions to the simply-typed lambda calculus and obtain a higher-order logic whose categorical semantics is captured by a topos [53]. We prove that every hereditarily domain-independent higher-order logic program can be translated into the nested relational algebra, thereby allowing higher-order logic to be used as query language and giving a direct, higher-order generalization of Codd's celebrated 1972 theorem establishing the equivalence of domain-independent first-order logic and relational algebra [22].
- *The type of finitely presented categories.* In chapter 4, we define FQL, the first query language for the functorial data model [74], demonstrate that the mappings between schemas form a functional query language, and build a compiler for FQL that emits SQL/PSM. In the functorial data model, database schemas are finitely presented categories, and every database instance over a schema C is a functor from C to the category of sets.

There are undoubtedly many other categorical types that may be usefully exploited by functional query languages, and we examine several possibilities in the conclusion. We have ordered the three chapters by expressive power: the nested relational calculus is a fragment of higher-order logic, which is a fragment of FQL. We conclude the introduction with a brief review of category theory.

Review of Category Theory

Category theory [9] is an axiomatically specified algebra of abstract functions suitable for formalizing mathematics. In contrast to traditional set theory, where structures are defined by what they *are*, in category theory structures are defined by how they *interact*. Compared to set theory, category theory is notable for its high level of abstraction and focus on compositionality. Since its inception in the 1940s, category theory has been applied in many disciplines, including information management, where categorical methods inspired the design of functional query languages such as the nested relational algebra [82]. The adjacent fields of mathematical logic and programming language theory also employ categorical techniques [53].

A *category* $\mathbf{C}$ consists of a class of *objects* $Ob(\mathbf{C})$ and a class of *morphisms* or *arrows* $Hom(\mathbf{C})$ between objects. Each morphism m has a source object S and a target object T, which we write as $m : S \to T$. Every object X has an identity morphism $id_X : X \to X$. When X is clear from the context we will write id_X as simply id. Two morphisms $f : B \to C$ and $g : A \to B$ may be *composed*, written $f \circ g : A \to C$ or $g; f : A \to C$. Composition is associative and id is its unit:

$$f \circ id = f \qquad id \circ f = f \qquad f \circ (g \circ h) = (f \circ g) \circ h$$

A morphism $f : X \to Y$ is an *isomorphism* when there exists a $g : Y \to X$ such that

$$f \circ g = id \qquad g \circ f = id$$

Two objects are *isomorphic* when there exists an an isomorphism between them.

Example categories include:

- **Set**, the category of sets. The objects of **Set** are sets, and a morphism $f : X \to Y$ is a (total) function from set X to set Y. Given morphisms $f : Y \to Z$ and $g : X \to Y$, the morphism $f \circ g : X \to Z$ is defined as function composition: $(f \circ g)(x) := f(g(x))$. The isomorphisms of **Set** are bijective functions. For each object X, id_X is the identity function on X.
- **Rel**, the category of relations. The objects of **Rel** are sets, and a morphism $f : X \to Y$ is a (partial) relation from set X to set Y. Given morphisms $f : Y \to Z$ and $g : X \to Y$, the morphism $f \circ g : X \to Z$ is defined as relation composition: $(f \circ g)(x, z) := \exists y \in Y.g(x, y) \wedge f(y, z)$. Like **Set**, the isomorphisms of **Rel** are bijective functions and identities are identity functions.
- Any preordered set $(P, \leq)$ is a category, where the objects are the members of P, and there is a morphism from X to Y exactly when $X \leq Y$. Between any two objects there can be at most one morphism, and the existence of identity morphisms and the composability of the morphisms are guaranteed by the reflexivity and the transitivity of $\leq$.
- Any directed graph generates a category, called the *free* category on the graph: its objects are the vertices of the graph, and the morphisms are the paths in the graph. For each vertex X, id_X is the 0-length path $X \to X$. Composition of morphisms is concatenation of paths, and there are no non-identity isomorphisms.

A *functor* $F : \mathbf{C} \to \mathbf{D}$ between two categories **C** and **D** is a mapping of objects of **C** to objects of **D** and morphisms of **C** to morphisms of **D** that preserves identities and composition:

$$F(f : X \to Y) : F(X) \to F(Y) \qquad F(id_X) = id_{F(X)} \qquad F(f \circ g) = F(f) \circ F(g)$$

Example functors include:

- For any category **C**, the identity functor $1_C : C \to C$ maps each object and morphism to itself.
- For any categories **C** and **D** and D an object of **D**, there exists a constant functor taking each object C in **C** to D and each morphism in C to id_D.
- There is a "forgetful" functor **Rel** $\to$ **Set** taking each function $f : X \to Y$ to its underlying relation $f \subseteq X \times Y$.

- The power set functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ maps each set to its power set and each function $f : X \to Y$ to the function which sends $U \subseteq X$ to its image $f(U) \subseteq Y$.
- For every set A, there is a functor $- \times A : \mathbf{Set} \to \mathbf{Set}$ that maps each set X to the cartesian product $X \times A$ and each function $f : X \to Y$ to the function $f \times id_A : X \times A \to Y \times A$.
- Consider preorders $(P, \leq)$ and $(Q, \preceq)$ as categories. A functor $F : P \to Q$ is just an order-preserving (monotone) function: if $a \leq b$ in P, then $F(a) \preceq F(b)$ in Q.
- Let K_n be the complete graph on n vertices that has an edge between every pair of distinct vertices. Then an n-coloring of a graph G is a functor $G \to K_n$.

A *natural transformation* $\alpha : F \Rightarrow G$ between two functors $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{C} \to \mathbf{D}$ is a family of morphisms $\alpha_X : F(X) \to G(X)$ in D, one for each object X in $\mathbf{C}$, such that for every $f : X \to Y$ in $\mathbf{C}$,

$$\alpha_Y \circ F(f) = G(f) \circ \alpha_X$$

This equation may conveniently expressed as a *commutative diagram*:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

A natural transformation α is a natural isomorphism when for every object X in $\mathbf{C}$, the morphism α_X is an isomorphism in $\mathbf{D}$. Example natural transformations include:

- The identity natural isomorphism $1_F : F \Rightarrow F$ for a functor $F : \mathbf{C} \to \mathbf{D}$ is defined as $1_{F_X} : F(X) \to F(X) := id_{F(X)}$.
- Consider the power set functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$. There is a natural transformation $sng : 1_{\mathbf{Set}} \Rightarrow \mathcal{P}$ that maps every set X to the singleton set $\{X\}$ (i.e., $sng_X : X \to \mathcal{P}(X)$), and there is a natural transformation $union : \mathcal{P} \circ \mathcal{P} \Rightarrow \mathcal{P}$ that maps a set of sets $\{X_1, \ldots, X_n\}$ to its n-ary union $X_1 \cup \ldots \cup X_n$ (i.e., $union_X : \mathcal{P}(\mathcal{P}(X)) \to \mathcal{P}(X)$).
- Let A be a set and consider the product functor $- \times A : \mathbf{Set} \to \mathbf{Set}$. There is a natural transformation $proj : - \times A \Rightarrow 1_{\mathbf{Set}}$ that for each set X maps the cartesian product $X \times A$ to its projection X (i.e., $proj_X : X \times A \to X$).

An *adjunction* between categories $\mathbf{C}$ and $\mathbf{D}$ consists of a functor $F : \mathbf{D} \to \mathbf{C}$ called the left adjoint, a functor $G : \mathbf{C} \to \mathbf{D}$ called the right adjoint, a natural transformation $\epsilon : F \circ G \Rightarrow 1_{\mathbf{C}}$ called the counit, and a natural transformation $\eta : 1_{\mathbf{D}} \Rightarrow G \circ F$ called the unit, such that for every object X in $\mathbf{C}$ and Y in $\mathbf{D}$, the following equations hold:

$$id_{F(Y)} = \epsilon_{F(Y)} \circ F(\eta_Y) \qquad id_{G(X)} = G(\epsilon_X) \circ \eta_{G(X)}$$

Consequently, the set of morphisms $F(Y) \to X$ is bijective with the set of morphisms $Y \to G(X)$. Example adjunctions include:

- Let $(P, \leq)$ and $(Q, \preceq)$ be partially ordered sets considered as categories, and let $F : Q \to P$ and $G : P \to Q$ be monotone functions considered as functors. If we have $F(Y) \leq X$ iff $Y \preceq G(X)$, then F and G are adjoint.
- Let A be a set and consider the product functor $- \times A : \mathbf{Set} \to \mathbf{Set}$. The exponential functor $-^A : \mathbf{Set} \to \mathbf{Set}$, which maps each set X to the set of functions from A to X (written X^A), is right adjoint to $- \times A$. Intuitively, this is because the set of functions $X \times Y \to Z$ is bijective with the set of functions $X \to Z^Y$.
- Consider the category of groups and group homomorphisms, $\mathbf{Grp}$. The functor $free : \mathbf{Set} \to \mathbf{Grp}$, which maps each set X to the free group generated by X, and the functor $forget : \mathbf{Grp} \to \mathbf{Set}$ which maps each group to its underlying set, are adjoint. Intuitively, maps from the free group $free(X)$ to a group Y correspond precisely to maps from the set X to the set $forget(Y)$: each homomorphism from $free(X)$ to Y is fully determined by its action on generators.

A *monad* on a category $\mathbf{C}$ consists of a functor $T : \mathbf{C} \to \mathbf{C}$ together with two natural transformations $\eta : 1_{\mathbf{C}} \Rightarrow T$ (called the unit) and $\mu : T \circ T \Rightarrow T$ (called the join) such that

$$\mu_X \circ T(\mu_X) = \mu_X \circ \mu_{T(X)} \qquad \mu_X \circ T(\eta_X) = \mu_X \circ \eta_{T(X)} = T(X)$$

Example monads include:

- If F and G are adjoint functors, with F left adjoint to G, then $G \circ F$ is a monad.
- If $(P, \leq)$ is a partially ordered set considered as a category, a functor $cl : P \to P$ is a monad exactly when $x \leq cl(y) \leftrightarrow cl(x) \leq cl(y)$ for all x, y in P.
- The power set functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ is a monad.

Chapter 2

Reifying Database Integrity Constraints as Identity Types

2.1 Introduction

In this chapter we describe how embedded dependencies [2], which are a widely used class of data integrity constraint, can be reified as identity types in dependently-typed monadic functional query languages, and how the well-known "chase" optimization procedure, which minimizes monad comprehensions in the presence of embedded dependencies [2], is sound in such languages. The chapter is split into two parts. In the first part, we discuss monad comprehensions, embedded dependencies, the chase procedure, and how the chase optimizes monad comprehensions in the presence of embedded dependencies. In the second part, we prove that the chase is a semantics preserving re-writing procedure in the context of Coq [11].

2.2 Semantic Optimization

Languages and systems such as MapReduceMerge [84], Ferry [42], Data Parallel Haskell [19], DyadLINQ [50], PIG [67], Fortress [4] and SciDB [25] are proliferating as Moore's law drives the cost of computing ever lower and the size of data ever larger. Like their predecessors SQL, NESL [14], and Kleisli [83], these declarative, collection-oriented languages and systems lift programming to the level of

abstract collections such as sets, bags, lists, and trees. As the database community discovered long ago, the sheer size of the data processed by these systems demands sophisticated optimization [54]. Simply choosing the right order to iterate over several collections can mean the difference between a query that completes in a few seconds instead of a few days. At this scale, the particular properties of the data become important [37].

Although these languages vary in the kinds of queries and collections they support, large fragments of these languages can be formalized in a uniform way using *monads* (to model collections) and *comprehensions* (to model queries) [17, 41]. Although monads have seen great success in providing structure to functional programs [80], sophisticated reasoning about monads using a priori semantic information has traditionally belonged to the realm of database theory. For example, in relational query processing, data integrity constraints capture such semantic information as keys, functional dependencies, inclusions, and join decompositions. These constraints are used as additional rewrite rules during optimization, a process known as *semantic optimization* [2, 28, 70].

For example [2], consider the following contrived query over a relation (set of records) $Movies$ with fields title, $\mathsf{director}$, and actor:

$$\begin{aligned}&\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies)\\&\texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title}\\&\texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})\end{aligned}$$

This query returns (a set of) tuples (d, a) where a acted in a movie directed by d. A naive implementation of this query will require a join. However, when $Movies$ satisfies the the functional dependency $\mathsf{title} \to \mathsf{director}$ (meaning that if $(\mathsf{director} : d, \mathsf{title} : t, \mathsf{actor} : a)$ and $(\mathsf{director} : d', \mathsf{title} : t', \mathsf{actor} : a')$ are $Movies$ records such that $t = t'$, then $d = d'$), this query is equivalent to

$$\begin{aligned}&\texttt{for}\ (m\ \texttt{in}\ Movies)\\&\texttt{return}\ (m.\mathsf{director}, m.\mathsf{actor})\end{aligned}$$

which can be evaluated without a join. (Note that if $Movies$ did not satisfy the functional dependency, the equivalence would not necessarily hold.)

Of course, knowing that the functional dependency holds, a programmer might simply write the optimized query to begin with. But constraints are not always known at compile time, such as when collections are indexed on-the-fly. Moreover, people are not always the programmers: information integration systems such as Midas [7] and Clio [44] automatically generate large amounts of code. The significant, potentially order-of-magnitude speed-ups enabled by semantic optimization are well-documented in the literature and applied in commercial databases, such as DB2 [46]. One of our goals in this chapter is to introduce semantic optimization to programming languages more generally.

Our `for − where − return` notation is defined in terms of an arbitrary monad, and the soundness of semantic optimization varies from monad to monad. For example, the semantic optimization procedure described in this chapter is not sound for lists or bags. Nevertheless, we see semantic optimization as useful not only for large-scale collection processing, but for other computations that can be modeled, at least in part, using monad comprehensions, such as functional-logic programming in Curry [5] and Daedalus [45], as well as probabilistic programming in Haskell [29] and IBAL [68].

Related Work

Semantic optimization (which conditionally preserves semantics, subject to constraints) complements non-semantic optimization (which always preserves semantics). Relational algebra has a well-developed theory of non-semantic optimization by minimizing detailed cost models [37], and cost models for monad comprehensions have been developed [51]. Inductive datatypes (and function types [59]) and monads as found in functional programming have a well-developed theory of non-semantic optimization by fold-fusion and deforestation [13, 39, 41, 49, 58, 60]. More recently, practical advances in theorem proving have sparked renewed interest in the duality between program verification and semantic optimization [47].

2.3 Monads

Monads are defined formally using category theory in the introduction to this dissertation, but for the purposes of this chapter we will use "monads in the functional programming style" [65]. In functional programming, a monad consists of a type-constructor M and two operations, $\texttt{return} : t \to M\ t$ and $\texttt{bind} : M\ t \to (t \to M\ t') \to M\ t'$, such that the following three laws hold:

$$\texttt{bind}\ (\texttt{return}\ x)\ f = fx \qquad \texttt{bind}\ m\ \texttt{return} = m \qquad \texttt{bind}\ (\texttt{bind}\ m\ f)\ g = \texttt{bind}\ m\ (\lambda x.\ \texttt{bind}\ (fx)\ g)$$

A monad with zero has another operation, $\texttt{zero} : M\ t$, such that two additional laws hold:

$$\texttt{bind zero}\ f\ =\ \texttt{zero} \qquad \texttt{bind}\ m\ (\lambda x.\ \texttt{zero})\ =\ \texttt{zero}$$

2.3.1 Examples

Monads with zeros are often used to model collections. For example, consider lists and sets in Haskell, in so-called "insert presentation":

```
data Ins a = Nil | Cons a (Ins a)


-- list monad
instance MonadZero Ins where
 return x = Cons x Nil
 bind x f = append (map f x)
 zero = Nil


-- set monad
instance MonadZero Ins where
 return x = Cons x Nil
 bind x f = union (map f x)
 zero = Nil
```

Monads are not tied to particular presentations. For example, the list and set monad can also be defined using so-called "union presentation":

```
data Un = Empty | Singleton a | Union (Un a) (Un a)
```

Not all collections have zeroes—for example, binary trees with non-empty leaves do not have a zero.

Monads can also be formed from functions; here, state with exceptions forms a monad with zero:

```
type ST s a = (s -> Maybe (a, s))


instance MonadZero (ST Int) where
 return s a = Just (s, a)
```

```
bind c f s = case c s of
                Nothing -> Nothing
                Just (s', a') -> f a' s'
zero = \s -> Nothing
```

Monads are by now an important subject in their own right. We refer the reader to [41, 80] for more details.

2.3.2 Notation

Monads are often used with so-called *do-notation*, which in Haskell looks like:

```
do x <- m1
   m2
=
bind m1 (\x -> m2)
```

Haskell programmers typically first encounter do-notation with Haskell's IO monad, as in the following program which outputs "Hello World":

```
main = do putStr "Hello"
          putStrLn "World"
```

Also popular is *monad comprehension notation*, which works for monads with zero, such as lists and sets:

```
[c | x <- X, P]
=
do x <- X
   if P then return c else zero
```

For example,

```
[x | x <- 1..10 , isEven x] = [2, 4, 6, 8, 10]
```

To emphasize the connection with database theory, we will use `for` − `where` − `return` notation, which we define in the next section. Regardless of the choice of notation, monad comprehensions can be normalized using the monad laws, as described by Grust in [41]. An interesting direction for future work would be to use a weaker structure, such as applicative functors [57], in place of monads in our theory.

2.4 Queries

We will be focusing on comprehensions that are syntactically *conjunctive queries*. For ease of exposition, in this section, we will assume we are working in a strongly-normalizing typed λ-calculus with first-class records, such as [38]. We will write $(l_1 : e_1, \ldots, l_N : e_N)$ to indicate a record with labels $l_1, \ldots l_N$ formed from expressions $e_1, \ldots, e_N$. We will assume records contain unique labels and are equated up-to label permutation. We also assume a decidable equality on records and sets of records. For the most part, in this section the specifics of our ambient language will not matter. We will abbreviate (potentially 0-length) vectors of variables $x_1, ..., x_N$ as $\overrightarrow{x}$. Fix a monad with zero M and let $\overrightarrow{X : M\ t}$ in some typing context. We will write $P(\overrightarrow{x})$ to indicate a conjunction of predicates over the variables $\overrightarrow{x}$. A *tableau* (plural: tableaux) has the form:

$$\begin{array}{l}\texttt{for } \overrightarrow{(x \texttt{ in } X)} \\ \texttt{where } P(\overrightarrow{x})\end{array}$$

The $\overrightarrow{(x \texttt{ in } X)}$ are called *generators*, and the $\overrightarrow{X}$ are called *roots*. A *query* is a pair of a tableau and an expression $R(\overrightarrow{x})$:

$$\begin{array}{l}\texttt{for } \overrightarrow{(x \texttt{ in } X)} \\ \texttt{where } P(\overrightarrow{x}) \\ \texttt{return } R(\overrightarrow{x})\end{array}$$

A query is interpreted as a monad comprehension:

$$\begin{array}{ll}\texttt{do} & x_1 \leftarrow X_1 \\ & \ldots \\ & x_N \leftarrow X_N \\ & \texttt{if } P(x_1, \ldots, x_N) \\ & \texttt{then return } R(x_1, \ldots, x_N) \\ & \texttt{else zero}\end{array}$$

For example, the query from the introduction:

$$
\begin{array}{l}
\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies) \\
\texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\
\texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})
\end{array}
$$

is interpreted as:

$$
\begin{array}{ll}
\texttt{do} & m_1 \leftarrow Movies \\
& m_2 \leftarrow Movies \\
& \texttt{if}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\
& \texttt{then return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \\
& \texttt{else zero}
\end{array}
$$

which de-sugars into

$$
\begin{array}{l}
\texttt{bind}\ Movies\ (\lambda m_1. \\
\ \texttt{bind}\ Movies\ (\lambda m_2. \\
\ \ \texttt{if}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\
\ \ \texttt{then return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \\
\ \ \texttt{else zero}\ ))
\end{array}
$$

and, in the set monad, this becomes

$$
\begin{array}{l}
\texttt{union}\ (\texttt{map}\ Movies\ (\lambda m_1. \\
\ (\texttt{union}\ (\texttt{map}\ Movies\ (\lambda m_2. \\
\ \ \texttt{if}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\
\ \ \texttt{then Cons}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})\ \texttt{Nil} \\
\ \ \texttt{else Nil}\ )))))
\end{array}
$$

A query can also naturally be interpreted as a function over its roots (here, $Movies$). In this case, to evaluate a query we require values for the roots (here, we require a particular relation $Movies$). We will write $q(I)$ to indicate a query q evaluated at I. The I is usually called an *instance*. Our example query can thus also be regarded as the function:

$$\begin{array}{lll} \lambda Movies. & \texttt{do} & m_1 \leftarrow Movies \\ & & m_2 \leftarrow Movies \\ & & \texttt{if } m_1.\mathsf{title} = m_2.\mathsf{title} \\ & & \texttt{then return } (m_1.\mathsf{director}, m_2.\mathsf{actor}) \\ & & \texttt{else zero} \end{array}$$

Extensions

Many extensions to conjunctive queries have been studied in the literature. Two stand out as particularly important:

- It is possible to allow generators to be dependent; for example:

  $$\texttt{for } (g \texttt{ in } Groups)\ (person \texttt{ in } g)\ \ldots$$

  This allows for nested values; for example, nested relations [70].

- It is possible to interpret queries in *monad algebras*, rather than monads [54]. A monad algebra is an operation of type $M\ t \to (t \to t') \to t'$ obeying certain equations. This more general type (relative to `bind`) allows for aggregation operations; for example, it is possible to write a query to count the number of elements in a list, which is impossible in the system presented above.

We will ignore these extensions for now, but it is likely that our results will hold in these more general settings (such as the nested relational calculus studied in the next chapter).

2.5 Embedded Dependencies

Embedded dependencies [2] take the form of pairs of tableaux. Intuitively, one tableaux is universally quantified, and the other existentially:

$$\begin{array}{l}\texttt{forall } \overrightarrow{(x \texttt{ in } X)} \\ \texttt{where } P(\overrightarrow{x}) \\ \texttt{exists } \overrightarrow{(y \texttt{ in } Y)} \\ \texttt{where } B(\overrightarrow{x}, \overrightarrow{y})\end{array}$$

The functional dependency from our example is written (the `exists` clause is empty):

$$\begin{array}{l}\texttt{forall } (x \texttt{ in } Movies)\ (y \texttt{ in } Movies) \\ \texttt{where } x.\mathsf{title} = y.\mathsf{title}, \\ \texttt{exists} \\ \texttt{where } x.\mathsf{director} = y.\mathsf{director}\end{array}$$

Unlike conjunctive queries, which have a straightforward interpretation in a monad with zero, the meaning of an embedded dependency is less clear. We will give the meaning of an embedded dependency C using a pair of queries called the *front* and *back* of C. We write $\mathcal{L}(\overrightarrow{x})$ to indicate a record capturing the variables $\overrightarrow{x}$; e.g., $(\mathsf{x}_1 : x_1, \ldots, \mathsf{x}_N : x_N)$. The front of an embedded dependency is:

$$\begin{array}{l}\texttt{for } \overrightarrow{(x \texttt{ in } X)} \\ \texttt{where } P(\overrightarrow{x}) \\ \texttt{return } \mathcal{L}(\overrightarrow{x})\end{array}$$

and the back is

$$\begin{array}{l}\texttt{for } \overrightarrow{(x \texttt{ in } X)}\ \overrightarrow{(y \texttt{ in } Y)} \\ \texttt{where } P(\overrightarrow{x}) \wedge B(\overrightarrow{x}, \overrightarrow{y}) \\ \texttt{return } \mathcal{L}(\overrightarrow{x})\end{array}$$

Later, we will write $front(R, C)$ and $back(R, C)$ to indicate the queries $front(C)$ and $back(C)$ but whose `return` clauses are R. We will write $I \models C$ to indicate that C holds of instance I, or that I *satisfies* C:

$$I \models C \quad := \quad front(C)(I) = back(C)(I)$$

In the set monad, the above definition of satisfaction corresponds to our intuitive notion of satisfaction; however, this definition of satisfaction has the advantage of being definable for every monad with zero.

Continuing with our example, our functional dependency holds of a particular instance $Movies$ when

$$\begin{array}{l}
\texttt{for}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies) \\
\texttt{where}\ x.\mathsf{title} = y.\mathsf{title}, \\
\texttt{return}\ (\mathsf{x} : x, \mathsf{y} : y) \\
= \\
\texttt{for}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies) \\
\texttt{where}\ x.\mathsf{title} = y.\mathsf{title} \wedge x.\mathsf{director} = y.\mathsf{director} \\
\texttt{return}\ (\mathsf{x} : x, \mathsf{y} : y)
\end{array}$$

For example, in this instance:

title	director	actor
T	D	A
T	D	B

the constraint holds because both sides evaluate to (omitting some labels to save space):

x	y
(T, D, A)	(T, D, A)
(T, D, A)	(T, D, B)
(T, D, B)	(T, D, A)
(T, D, B)	(T, D, B)

whereas in this instance:

title	director	actor
T	D_1	A
T	D_2	B

the constraint does not hold because the left-hand side and right-hand side evaluate to, respectively:

x	y
(T, D_1, A)	(T, D_1, A)
(T, D_1, A)	(T, D_2, B)
(T, D_2, B)	(T, D_1, A)
(T, D_2, B)	(T, D_2, B)

x	y
(T, D_1, A)	(T, D_1, A)
(T, D_2, B)	(T, D_2, B)

2.6 The Chase

The chase is a confluent rewriting system that rewrites comprehensions using embedded dependencies [2]. We now describe the chase, and in the next section we show how to use it to optimize queries.

2.6.1 Homomorphisms

A homomorphism between queries, $h : Q_1 \to Q_2$

$$
\begin{array}{rcl}
Q_1 & := & \texttt{for } \overrightarrow{(v_1 \texttt{ in } V_1)} \\
 & & \texttt{where } P_1(\overrightarrow{v_1}) \\
 & & \texttt{return } R_1(\overrightarrow{v_1}) \\
\to_h & & \\
Q_2 & := & \texttt{for } \overrightarrow{(v_2 \texttt{ in } V_2)} \\
 & & \texttt{where } P_2(\overrightarrow{v_2}) \\
 & & \texttt{return } R_2(\overrightarrow{v_2})
\end{array}
$$

is a substitution mapping the `for` -bound variables of Q_1 (namely, $\overrightarrow{v_1}$) to the `for` -bound variables of Q_2 (namely, $\overrightarrow{v_2}$) that preserves the structure of Q_1 in the sense that

- Each $(h(v_{1_i})$ `in` $V_{1_i})$ appears in $\overrightarrow{(v_2 \texttt{ in } V_2)}$ (that is, the image of each generator in Q_1 is found in the generators of Q_2).
- $P_1(h(\overrightarrow{v_1}))$ is entailed by $P_2(\overrightarrow{v_2})$ (that is, the images of the conjuncts in Q_1 are a consequence of the conjuncts in Q_2).
- $R_1(h(\overrightarrow{v_1})) = R_2(\overrightarrow{v_2})$, under the equalities in P_2 (that is, the `return` clauses are equivalent).

For arbitrary predicates P_1 and P_2 and arbitrary expressions R_1 and R_2, finding homomorphisms is undecidable. However, when the queries are *path-conjunctive*—that is, when P_1, P_2 are conjunctions of equalities between paths of the form $v.l_1, \ldots l_n$, and R_1 and R_2 are records built from paths—finding homomorphisms is decidable but NP-hard. Moreover, in this case there are practical, sound heuristics [28] based on pruning the search space of substitutions to remove candidates that are "obviously wrong" based on a partial variable assignment. In this chapter, all our examples are path conjunctive.

For example, consider our *Movies* query (call it Q_1):

$$\begin{aligned} Q_1 \quad := \quad & \texttt{for } (m_1 \texttt{ in } \mathit{Movies}) \ (m_2 \texttt{ in } \mathit{Movies}) \\ & \texttt{where } m_1.\mathsf{title} = m_2.\mathsf{title} \\ & \texttt{return } (m_1.\mathsf{director}, m_2.\mathsf{actor}) \end{aligned}$$

and also the normalized query (call it Q_2) which we will later optimize Q_1 into:

$$\begin{aligned} Q_2 \quad := \quad & \texttt{for } (m \texttt{ in } \mathit{Movies}) \\ & \texttt{return } (m.\mathsf{director}, m.\mathsf{actor}) \end{aligned}$$

There is a homomorphism $h : Q_1 \to Q_2$; namely, the substitution $m_1 \mapsto m, m_2 \mapsto m$. To check this, we first apply h to Q_1:

$$\begin{aligned} h(Q_1) \quad := \quad & \texttt{for } (m \texttt{ in } \mathit{Movies}) \ (m \texttt{ in } \mathit{Movies}) \\ & \texttt{where } m.\mathsf{title} = m.\mathsf{title} \\ & \texttt{return } (m.\mathsf{director}, m.\mathsf{actor}) \end{aligned}$$

In $h(Q_1)$ each generator (m `in` $Movies$) appears in Q_2. Moreover, the `where` clause of $h(Q_1)$ is a tautology and hence is entailed by the (empty) `where` clause of Q_2. Finally, the two `return` clauses are equal. As such, the substitution $m_1 \mapsto m, m_2 \mapsto m$ is a homomorphism.

In the set monad, homomorphisms are useful because the existence of a homomorphism $A \to B$ implies that for every I, $B(I) \subseteq A(I)$. Indeed, it is easy to see in this example that $Q_2(I) \subseteq Q_1(I)$ for any I. Later we will make use of a similar property for arbitrary monads to show that the chase is sound.

At this point it is instructive to check that there is no homomorphism $Q_2 \to Q_1$. There are only two candidate substitutions: $m \mapsto m_1$ and $m \mapsto m_2$. Neither of these works because neither of the images of Q_2's `return` clause (neither `return` (m_1.director, m_1.actor) nor `return` (m_2.director, m_2.actor)) is equivalent to Q_1's `return` clause (`return` (m_1.director, m_2.actor)), even under the equality in Q_1 (m_1.title $= m_2$.title). Because there are not homomorphisms in both directions, these two queries are not equivalent. Indeed, consider the instance:

title	director	actor
T	D_1	A
T	D_2	B

Q_1 and Q_2 evaluate to, respectively

director	actor
D_1	A
D_1	B
D_2	A
D_2	B

director	actor
D_1	A
D_2	B

Of course, if we had chosen an instance I that satisfied the functional dependency Title $\to$ Director, then $Q_1(I)$ and $Q_2(I)$ would have evaluated to the same result.

2.6.2 The Chase Algorithm

Now we can define *the chase*. Let

$$
\begin{array}{rcl}
C & := & \texttt{forall } \overrightarrow{(x \texttt{ in } X)} \\
 & & \texttt{where } P(\overrightarrow{x}) \\
 & & \texttt{exists } \overrightarrow{(y \texttt{ in } Y)} \\
 & & \texttt{where } B(\overrightarrow{x}, \overrightarrow{y})
\end{array}
\qquad\qquad
\begin{array}{rcl}
Q & := & \texttt{for } \overrightarrow{(v \texttt{ in } V)} \\
 & & \texttt{where } O(\overrightarrow{v}) \\
 & & \texttt{return } R(\overrightarrow{v})
\end{array}
$$

and suppose there exists a homomorphism $h : front(R, C) \to Q$. Then a *chase step* is to rewrite Q into $chase(Q, C)$ by adding the image of the existential part of C:

$$
\begin{array}{rcl}
chase(Q, C) & := & \texttt{for } \overrightarrow{(v \texttt{ in } V)}\ \overrightarrow{(y \texttt{ in } Y)} \\
 & & \texttt{where } O(\overrightarrow{v}) \wedge B(\overrightarrow{h(x)}, \overrightarrow{y}) \\
 & & \texttt{return } R(\overrightarrow{v})
\end{array}
$$

The chase itself is to repeatedly rewrite Q by looking for homomorphisms from C:

$$Q \rightsquigarrow chase(Q, C) \rightsquigarrow chase(chase(Q, C), C) \rightsquigarrow \ldots$$

Termination of the chase is undecidable, but if it terminates it will converge to a unique fixed point [28] provided that we do not take a chase step when there is a homomorphism extending h from $chase(Q, C)$ to Q. Continuing with our $Movies$ example, we can see that there is a homomorphism $x \mapsto m_1, y \mapsto m_2$ from the front of our constraint:

$$
\begin{array}{l}
\texttt{forall } (x \texttt{ in } Movies)\ (y \texttt{ in } Movies) \\
\texttt{where } x.\mathsf{title} = y.\mathsf{title}, \\
\texttt{exists} \\
\texttt{where } x.\mathsf{director} = y.\mathsf{director}
\end{array}
$$

to our original query:

$$\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies)$$
$$\texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title}$$
$$\texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})$$

Hence, the chase applies, and $chase(Q, C)$ is:

$$\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies)$$
$$\texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \wedge m_1.\mathsf{director} = m_2.\mathsf{director}$$
$$\texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})$$

At this point, we stop chasing, because we have that $chase(chase(Q, C), C) = chase(Q, C)$ and hence there is a homomorphism $chase(chase(Q, C), C) \to chase(Q, C)$. In general, it is not enough to check for the syntactic equality of $chase(Q, C)$ and Q to stop the chase, as queries can be equivalent without being syntactically equal. Hence, we must use homomorphisms to detect termination.

2.6.3 Soundness

The chase is not sound for arbitrary monads, and in particular it is not sound for the list and bag monads [70]. The chase adds generators to a query, and adding generators to list and bag comprehensions can add additional tuples to the result; in the set monad, these extra tuples disappear by idempotency.

For example, in the list monad, our functional dependency $\mathsf{title} \to \mathsf{director}$ still holds on this instance:

title	director	actor
T	D	A
T	D	B

but our original and optimized queries are not equivalent; they result in, respectively,

$D \quad A$

$D \quad B$

$D \quad A$

$D \quad B$

$D \quad A$

$D \quad B$

In a tech report [81] we give sufficient conditions on a monad for the chase to be sound. As later we will prove that the chase is sound when specialized to Coq's ensemble monad, we omit the proof here. However, for reference we state here what it means for the chase to be sound and give a broad outline of the steps required to prove it. Let

$$\begin{aligned} Q \quad := \quad & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\ & \texttt{where } C(\overrightarrow{x}) \\ & \texttt{return } E(\overrightarrow{x}) \end{aligned}$$

$$\begin{aligned} d \quad := \quad & \texttt{forall } \overrightarrow{(r \texttt{ in } R)} \\ & \texttt{where } B_1(\overrightarrow{r}) \\ & \texttt{exists } \overrightarrow{(s \texttt{ in } S)} \\ & \texttt{where } B_2(\overrightarrow{r}, \overrightarrow{s}) \end{aligned}$$

Let $h : front(d) \to Q$. The chase is sound when forall I s.t. $I \models d$, $Q(I) = Q'(I)$, where

$$\begin{aligned} Q' \quad := \quad & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(s \texttt{ in } S)} \\ & \texttt{where } C(\overrightarrow{x}) \wedge B_2(h(\overrightarrow{r}), \overrightarrow{s}) \\ & \texttt{return } E(\overrightarrow{x}) \end{aligned}$$

The proof proceeds along five steps, where steps 4 and 5 are symmetrical to 1 and 2.

$$
\begin{aligned}
Q \quad &:= \quad \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\
&\qquad \texttt{where } C(\overrightarrow{x}) \\
&\qquad \texttt{return } E(\overrightarrow{x}) \\
C(\overrightarrow{x}) \vdash B_1(h(\overrightarrow{r})) \quad &= \quad (1) \\
&\qquad \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\
&\qquad \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \\
&\qquad \texttt{return } E(\overrightarrow{x}) \\
h(\overrightarrow{r \texttt{ in } R}) \subseteq \overrightarrow{x \texttt{ in } P} \quad &= \quad (2) \\
&\qquad \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)} \\
&\qquad \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\
&\qquad \texttt{return } E(\overrightarrow{x}) \\
d \text{ holds} \quad &= \quad (3) \\
&\qquad \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)}\ \overrightarrow{(s \texttt{ in } S)} \\
&\qquad \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge B_2(\overrightarrow{v}, \overrightarrow{s}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\
&\qquad \texttt{return } E(\overrightarrow{x}) \\
h(\overrightarrow{r \texttt{ in } R}) \subseteq \overrightarrow{x \texttt{ in } P} \quad &= \quad (4) \\
&\qquad \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(s \texttt{ in } S)} \\
&\qquad \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \wedge B_2(h(\overrightarrow{r}), \overrightarrow{s}) \\
&\qquad \texttt{return } E(\overrightarrow{x}) \\
C(\overrightarrow{x}) \vdash B_1(h(\overrightarrow{r})) \quad &= \quad (5) \\
Q' \quad &:= \quad \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(s \texttt{ in } S)} \\
&\qquad \texttt{where } C(\overrightarrow{x}) \wedge B_2(h(\overrightarrow{r}), \overrightarrow{s}) \\
&\qquad \texttt{return } E(\overrightarrow{x})
\end{aligned}
$$

2.7 Tableaux Minimization

We now demonstrate how to minimize queries in the presence of constraints, following Popa et al [28]. The soundness of this procedure follows from the soundness of the chase. Suppose we are given a query Q and constraints C. We first chase Q with C to obtain U, a so-called *universal plan*. We then search for subqueries of U (obtained by removing generators from U), chasing each in turn with C to check for equivalence with U.

2.7.1 Example - Movies

Start with:

$$\begin{aligned} Q \quad := \quad & \texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies) \\ & \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\ & \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \end{aligned}$$

$$\begin{aligned} C \quad := \quad & \texttt{for}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies) \\ & \texttt{where}\ x.\mathsf{title} = y.\mathsf{title} \\ & x.\mathsf{director} = y.\mathsf{director} \end{aligned}$$

The chased query—the universal plan—is:

$$\begin{aligned} U \quad := \quad & \texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies) \\ & \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \wedge m_1.\mathsf{director} = m_2.\mathsf{director} \\ & \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \end{aligned}$$

We may now proceed with tableau minimization by searching for subqueries of U. Removing the generator $(m_1\ \texttt{in}\ Movies)$ and replacing m_1 with m_2 in the body of the query gives a normalized query:

$$\begin{aligned} Q' \quad := \quad & \texttt{for}\ (m_2\ \texttt{in}\ Movies) \\ & \texttt{return}\ (m_2.\mathsf{director}, m_2.\mathsf{actor}) \end{aligned}$$

Now we look for a homomorphism $Q' \to U$. The identity substitution works; the important part here to notice is the `return` clause, where (m_2.director, m_2.actor) is equal to (m_1.director, m_2.actor) precisely because of the equality m_1.director = m_2.director, which appears in U but not in Q. Note that there is also a homomorphism $U \to Q'$ (namely, $m_2 \mapsto m, m_1 \mapsto m$); hence $U = Q' = Q$.

2.7.2 Example - Minimization without Constraints

Tableaux minimization can also be done *without* constraints. Indeed, this degenerate case was first proposed in 1977 [20]. Consider the (contrived) query:

```
for (x in X) (y in X)
where P(x)
return E(x)
```

This minimizes to the equivalent query:

```
for (z in X)
where P(z)
return E(z)
```

In the top-to-bottom direction, the homomorphism is $x \mapsto z, y \mapsto z$, and in the bottom-to-top direction the homomorphism is $z \mapsto x$.

2.7.3 Example - Indexing

We conclude this section with an optimization scenario involving a tuple-generating constraint (that is, a constraint with a non-empty `exists` clause). As we remarked in the introduction, a reasonably competent programmer might be able to optimize our *Movies* query directly, without applying the chase at all. But sometimes constraints are not available to the programmer, such as when indices are generated on the fly.

Consider the following query, which in the set monad returns the names of all *People* between 16 and 18 years old:

$$
\begin{array}{rcl}
Q & := & \texttt{for}\ (p\ \texttt{in}\ People) \\
 & & \texttt{where}\ p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18 \\
 & & \texttt{return}\ p.\mathsf{name}
\end{array}
$$

Depending on the underlying access patterns, or the whims of a database administrator, a modern relational database management system might transparently index *People* by creating another relation *Children*, such that the following constraint holds:

$$
\begin{array}{rcl}
C & := & \texttt{forall}\ (p\ \texttt{in}\ People) \\
 & & \texttt{where}\ p.\mathsf{age} < 21 \\
 & & \texttt{exists}\ (c\ \texttt{in}\ Children) \\
 & & \texttt{where}\ p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age}
\end{array}
$$

In order to effectively use this new relation, queries written against *People* must be rewritten, at runtime, to use *Children*. Tableaux minimization provides an automated mechanism to do so. First, we look for a homomorphism $front(C) \rightarrow Q$, and discover that the identity substitution works, because $p.\mathsf{age} < 21$ is entailed by $p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18$. Thus the chase applies and we obtain a universal plan:

$$
\begin{array}{rcl}
U & := & \texttt{for}\ (p\ \texttt{in}\ People)\ (c\ \texttt{in}\ Children) \\
 & & \texttt{where}\ p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18 \wedge \\
 & & \qquad p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \\
 & & \texttt{return}\ p.\mathsf{name}
\end{array}
$$

Now, we minimize the universal plan by removing the (p `in` $People$) generator (note that to do so we must replace each occurrence of p with some other well-typed variable, in this case c):

$$
\begin{aligned}
Q' \quad := \quad & \texttt{for}\ (c\ \texttt{in}\ Children) \\
& \texttt{where}\ c.\mathsf{age} > 16 \wedge c.\mathsf{age} < 18 \\
& \texttt{return}\ c.\mathsf{name}
\end{aligned}
$$

We check that $Q' = U$ by looking for homomorphisms in both directions. The identity substitution is a homomorphism $Q' \to U$, owing to the fact that $p.\mathsf{name} = c.\mathsf{name}$. But at this point there is no homomorphism $U \to Q'$, because there is no substitution h that makes ($h(p)$ `in` $People$) equal to (c `in` $Children$). In fact, C alone is not enough to prove that $Q' = Q$—there may be extra tuples in $Children$ that do not appear in $People$. But if our index was built correctly we know that an additional constraint holds:

$$
\begin{aligned}
C' \quad := \quad & \texttt{forall}\ (c\ \texttt{in}\ Children) \\
& \texttt{exists}\ (p\ \texttt{in}\ Person) \\
& \texttt{where}\ p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age}
\end{aligned}
$$

As such, we may chase Q' with C' (using the identity substitution) to obtain the equivalent:

$$
\begin{aligned}
Q'' \quad := \quad & \texttt{for}\ (c\ \texttt{in}\ Children)\ (p\ \texttt{in}\ Person) \\
& \texttt{where}\ c.\mathsf{age} > 16 \wedge c.\mathsf{age} < 18 \wedge \\
& \qquad p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \\
& \texttt{return}\ c.\mathsf{name}
\end{aligned}
$$

Now we can see that the identity substitution is a homomorphism $Q'' \to U$ (again owing to the fact that $p.\mathsf{name} = c.\mathsf{name}$). We have thus concluded that $Q'' = Q' = Q = U$.

2.8 Embedded Dependencies in Coq

Because an embedded dependency d can be represented as the equivalence of two conjunctive queries, $front(d)$ and $back(d)$, it is simple to reify embedded dependencies in dependent type theories like Coq [11]. All the Coq code in this section is available online at wisnesky.net/chase.v. Consider the set monad, defined for expediency in Coq as an "ensemble" (similar to how sets are encoded in higher-order logic, as we will see in the next chapter):

```
Definition set t := t -> Prop.

Definition zero {t} : set t := fun x:t => False.

Definition plus {t} (a b: set t) : set t := fun x:t => a x \/ b x.

Definition ret {t} (a: t) : set t := fun x:t => a = x.

Definition map {s t} (f : s -> t) (X : set s) : set t
 := fun y : t => exists x : s, X x /\ (f x = y).

Definition concat {t} (I : set (set t)) : set t :=
 fun j : t => exists J, I J /\ J j.

Definition bind {s t} (a: set s) (b: s -> set t) : set t :=
    concat (map b a).
```

In our movies example, the functional dependency

$$\texttt{forall } (x \texttt{ in } Movies) \ (y \texttt{ in } Movies)$$
$$\texttt{where } x.\text{title} = y.\text{title},$$
$$\texttt{exists}$$
$$\texttt{where } x.\text{director} = y.\text{director}$$

becomes the equivalence

$$
\begin{array}{rl}
 & \texttt{for}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies) \\
 & \texttt{where}\ x.\mathsf{title} = y.\mathsf{title}, \\
 & \texttt{return}\ (\mathsf{x} : x, \mathsf{y} : y) \\
= & \\
 & \texttt{for}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies) \\
 & \texttt{where}\ x.\mathsf{title} = y.\mathsf{title} \wedge x.\mathsf{director} = y.\mathsf{director} \\
 & \texttt{return}\ (\mathsf{x} : x, \mathsf{y} : y)
\end{array}
$$

which is rendered in Coq as

```
Record Movie := movie { title: string; director: string; actor:string }.


Definition fd (Movies: set Movie) : Prop :=
 bind Movies (fun x => bind Movies (fun y =>
  if string_dec (title x) (title y) then ret (x, y) else zero))
=
 bind Movies (fun x => bind Movies (fun y =>
  if and (string_dec (title x) (title y)) (string_dec (director x) (director y))
  then ret (x, y)  else zero)).
```

The key point is that because the functional dependency `fd` is a Coq proposition, programmers can manipulate proofs of it programatically. For example, we could write a program that only operates over instances for which the constraints holds:

```
Definition some_query (m: set Movie) (c: fd m) := ...
```

To use such a definition, the programmer can construct a proof that the constraint holds, for example with a singleton set

```
Definition ok_inst := fun m => ret (movie "T" "D" "A")


Theorem ok_inst_is_ok : fd ok_inst.
```

```
...

Definition some_query_on_ok_inst := some_query ok_inst ok_inst_is_ok
```

2.9 Soundness of the Chase in Coq

Reifying embedded dependencies as identity types not only allows programmers the ability to manipulate them as first class objects, but allows the chase procedure to apply as well. Unfortunately, because the chase is a *nominal* algorithm, making use of the concrete names of variables in queries and constraints, this process is difficult to capture in its full generality: a homomorphism between queries is a substitution mapping the bound variables of one query to another query, but Coq programmers cannot easily reify the names of bound variables as, say, strings.

Fortunately, it is possible to prove every *particular* chase sequence by following the proof described in [81]. Consider our general statement of chase soundness: let

$$
\begin{array}{rcl}
Q & := & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\
 & & \texttt{where } C(\overrightarrow{x}) \\
 & & \texttt{return } E(\overrightarrow{x})
\end{array}
\qquad\qquad
\begin{array}{rcl}
d & := & \texttt{forall } \overrightarrow{(r \texttt{ in } R)} \\
 & & \texttt{where } B_1(\overrightarrow{r}) \\
 & & \texttt{exists } \overrightarrow{(s \texttt{ in } S)} \\
 & & \texttt{where } B_2(\overrightarrow{r}, \overrightarrow{s})
\end{array}
$$

Let $h : front(d) \to Q$. The chase is sound when forall I s.t, $I \models d$, $Q(I) = Q'(I)$, where

$$
\begin{array}{rcl}
Q' & := & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(s \texttt{ in } S)} \\
 & & \texttt{where } C(\overrightarrow{x}) \wedge B_2(h(\overrightarrow{r}), \overrightarrow{s}) \\
 & & \texttt{return } E(\overrightarrow{x})
\end{array}
$$

To demonstrate what the proof of this looks like in Coq we will make two simplifications. First, we use vectors of length 1. Second, we make use of the homomorphism $front(d) \to Q$ by directly setting $r = x$ and $R = P$. It is precisely this identification of variables that is impossible to do in general, but is possible for each particular case. Then the result is provable in Coq as follows:

```
Theorem chase_sound {s t u}
  (P: set s)
  (C: s -> bool)
  (E: s -> t)
  (B1:s -> bool)
  (S: set u)
  (B2:s -> u -> bool)
  (d_holds : bind P (fun x => if B1 x then ret x else zero)
           = bind P (fun x => bind S (fun s =>
              if (B1 x) && (B2 x s) then ret x else zero)))
  (h : forall a, negb (C a) || (B1 a) = true) :
   bind P (fun x => if C x then ret (E x) else zero)
= bind P (fun x => bind S (fun s => if (C x) && (B2 x s) then ret (E x) else zero)).
```

Coq's tactic language Ltac might be able to automate the construction of chase proofs, but like Coq itself, Ltac cannot easily reify bound variable names. However, a Coq plug-in (written in ML) would be able to automatically construct chase proofs because it would have access to concrete variable names. Implementing such a plug-in is a promising direction for future work. Alternatively, a deep embedding of the monad language and dependencies would allow Ltac the required access to concrete variable names.

Chapter 3

Higher-order Logic as a Query Language

3.1 Introduction

In this chapter we describe how to use higher-order logic (HOL) [53] as a database query language. Our methodology is a higher-order generalization of Codd's pioneering work [22] on using first-order logic (FOL) as a database query language. Codd proved that FOL and the relational algebra (RA) have equal expressive power, thereby allowing many tools from mathematical logic to be brought to bear on problems in information management. Although SQL, which is based on RA rather than FOL, is the dominant user-facing programming language for relational database management systems, in other kinds of relational systems, such as information integration systems, FOL is the dominant language [31]. Indeed, most theoretical work on the relational model is done using FOL [1]. Similarly, most theoretical work on the nested relational model, which allows relations to be nested inside relations, is done using the nested relational calculus (NRC) [82], and the NRC is a fragment of HOL.

Our original motivation for studying HOL as a query language was to understand its unexpected appearance in two places. The first was the observation by Popa and Tannen [70] that the NRC, although inspired by the categorical notion of a monad, seems to have some of the structure of HOL. The second was the observation by Spivak [74] that the functorial data model, which we describe in the next chapter,

	Higher-order quantification	First-order quantification
Bounded quantification	Nested Relational Calculus	Relational Calculus
Unbounded quantification	Higher-order Logic	Set Theory

Figure 3.1: Summary of query calculi

also seems to have some of the structure of HOL. In fact, in both places what appeared was not HOL, but rather HOL's categorical semantics: the categorical notion of a *topos* [53]. Only after understanding the role that topoi played in both these works were we able to deduce the common theme: HOL was being used to query databases.

Since then we have discovered that HOL is an extremely expressive query calculus for the nested relational model. Unlike the NRC, which only allows bounded quantification over sets, HOL allows unbounded quantification over sets; unlike first-order set theory (FOST) [1], which treats sets as first-order objects, HOL treats sets as genuinely higher-order objects. Because most work on practical query optimization is done in an algebraic setting [41], both the NRC and FOST use translation into the nested relational algebra (NRA) as a primary implementation technique [1] [82]. Hence, we looked for a translation of HOL into the NRA, and that is the content of this chapter: a translation of HOL into the NRA and a corresponding semantics preservation proof.

3.1.1 Contributions

The technical contributions of this chapter are

- A categorical semantics for the NRC in boolean topoi [53].
- A translation from HOL to the NRC. By a result of Wong [82], the NRC can be translated to the NRA. Hence, we have a translation from HOL to the NRA.
- A categorical description of *domain independence* that generalizes existing notions [78]. Domain-independent queries are exactly those that do not depend on the underlying domain of the input database, be it 32-bit integers, ASCII strings, binary blobs, etc.
- A proof that our HOL to NRC translation is sound under HOL's set-theoretic semantics for hereditarily domain-independent terms. A HOL term is hereditarily domain-independent when it and all of its sub-terms are domain-independent.
- A mechanization of the semantics preservation proof in the Coq proof assistant [11].

We conjecture that our semantics preservation proof can be extended from hereditarily domain-independent terms to all domain-independent terms. A tool that translates arbitrary HOL to NRC is available at wisnesky.net/hol2nrc.jar. Our Coq proofs are available at wisnesky.net/hol2nrc.v.

3.1.2 Outline

This chapter is structured as follows:

- In section 2 we define the syntax and categorical semantics of HOL.
- In section 3 we define the syntax and categorical semantics of NRC.
- In section 4 we define a translation from HOL types to NRC types.
- In section 5 we define a categorical notion of change of domain.
- In section 6 we define NRC's active domain query.
- In section 7 we define a translation from HOL terms to NRC terms.
- In section 8 we define a notion of domain independence for HOL terms.
- In section 9 we prove our HOL to NRC translation is semantics preserving.
- In section 10 we provide a reverse translation from NRC to HOL.
- In section 11 we review related work.
- In section 12 we discuss future work.
- In section 13 we discuss our Coq mechanization of our semantics preservation proof.

Many of our proofs are found in this chapter's appendix. We conclude the introduction with a review of Codd's theorem [22].

3.1.3 Codd's Theorem

In this section we review Codd's theorem [22]. A *relational schema* is a set of relation names and their arities. An *instance* I over a schema $\mathbf{R}$ is a collection of relations I_R, one for each relation name $R \in \mathbf{R}$. A FOL formula P over schema $\mathbf{R}$ has the form:

$$P ::= \top \mid \bot \mid P \wedge P \mid P \vee P \mid \neg P \mid x_n = x_m \mid R(x_1, \ldots, x_n) \mid \forall x.P(x) \mid \exists x.P(x)$$

A *relational calculus* (RC) expression φ is a FOL formula with free variables:

$$\varphi ::= \{\ x_1, \ldots, x_n \mid P(x_1, \ldots, x_n)\ \}$$

A *model* of an RC expression is a pair $(\mathbf{D}, I)$, where $\mathbf{D}$ is a "domain" of constants over which variables are quantified and I is an instance such that for every R, I_R is a relation over $\mathbf{D}$ of appropriate arity. We write $q_{\mathbf{D}}(I)$ to indicate the result of evaluating RC expression q on instance I using domain $\mathbf{D}$.

A *relational algebra* expression E over schema $\mathbf{R}$ has the form, where $R \in \mathbf{R}$ and n represents a numeric column position:

$$E ::= R \mid \pi_{n_1,\ldots,n_k}\ E \mid \sigma_{n_1=n'_1,\ldots,n_j=n'_j}\ E \mid E \times E \mid E \cup E \mid E - E$$

Some RC expressions are clearly equivalent to RA expressions. For example, the RC expression $\{x \mid R(x)\}$ is clearly equivalent to the RA expression R. But not every RC expression is equivalent to an RA expression. Let q be

$$\{\ x_1, \ldots, x_n \mid \neg R(x_1, \ldots, x_n)\ \}$$

Intuitively, q produces different answers over different domains. If the variables $x_1, \ldots, x_n$ range over a domain $\mathbf{D}$, then $q_{\mathbf{D}}(I)$ evaluates to $\mathbf{D}^n - I_R$ for every I. Hence, this RC expression is not *domain-independent*. In contrast,

$$\{\ x_1, \ldots, x_n \mid R'(x_1, \ldots, x_n)\ \wedge\ \neg R(x_1, \ldots, x_n)\ \}$$

is domain-independent. More formally, the *active domain* of an instance I, written $adom(I)$, is the set of constants occurring in I. An RC expression q is *domain-independent* when for every I and $\mathbf{D}$ such that $adom(I) \subseteq \mathbf{D}$, we have that $q_{\mathbf{D}}(I) = q_{adom(I)}(I)$. Intuitively, for domain-independent queries q the result of evaluating $q_{\mathbf{D}}(I)$ does not depend on the domain $\mathbf{D}$, but only depends on the instance I.

Codd's theorem [22] is that we can translate from RC to RA by treating unbounded quantification, which has no RA counterpart, as quantification over the active domain, which does have an RA counterpart. For example, consider the RC expression $\{x \mid \forall y R(x, y)\}$. To translate it to RA we first convert it to the logically equivalent $\{x \mid \neg\exists y \neg R(x, y)\}$. Then we translate from RC to RA recursively:

$$adom := \pi_1(R) \cup \pi_2(R)$$

$$\neg R(x, y) := adom \times adom - R$$

$$\exists y \neg R(x, y) := \pi_1 \ (adom \times adom - R)$$

$$\neg \exists y \neg R(x, y) := adom - \pi_1 \ (adom \times adom - R)$$

Although there is an equivalent RA expression for every domain-independent RC expression, it is undecidable whether a given RC expression is domain-independent. However, there are various syntactic subsets of RC for which membership is decidable and for which all expressions are domain-independent [2]. Typically, RC expressions are simply assumed to be domain-independent, and "RC" is taken to mean "domain-independent FOL". Thus we can think of RC as FOL where every variable is implicitly quantified over the active domain.

3.2 Higher-order Logic

HOL is a family of related formalisms [53], and we use the following formulation. The set of HOL *types* t is inductively defined as:

$$\begin{array}{rcll} t & ::= & D & \text{domain type} \\ & | & 0 & \text{empty type} \\ & | & 1 & \text{unit type} \\ & | & t \times t & \text{pair type} \\ & | & t + t & \text{choice type} \\ & | & t \to 2 & \text{set (characteristic function) type} \end{array}$$

Intuitively, each type defines a set of values, called the *inhabitants* of that type: the inhabitants of D are constants, 0 has no inhabitants, 1 has a single inhabitant, the inhabitants of $t \times t'$ are pairs of inhabitants of t and t', the inhabitants of $t + t'$ are inhabitants of either t or t', and the inhabitants of $t \to 2$ are sets of inhabitants of t represented as characteristic functions. We define $2 := 1 + 1$ and treat it as a boolean type because it has two inhabitants.

The set of HOL *terms* e is inductively defined as those of a simply typed lambda calculus (e.g., [64]):

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
& | & \lambda x{:}t.e & \text{set comprehension} \\
& | & ee & \text{membership} \\
& | & () & \text{empty record} \\
& | & (e,e) & \text{pair} \\
& | & e.1 & \text{first projection of a pair} \\
& | & e.2 & \text{second projection of a pair} \\
& | & \mathit{ff}_t & \text{impossibility} \\
& | & \mathit{inl}_t\ e & \text{first injection into a choice} \\
& | & \mathit{inr}_t\ e & \text{second injection into a choice} \\
& | & \mathit{case}\ e\ \mathit{of}\ \lambda x{:}t.e\ \mathit{or}\ \lambda x{:}t.e & \text{conditional} \\
& | & e = e & \text{equality test}
\end{array}
$$

As usual, we assume all variables in a term are unique, and equate terms that differ only by bound variables. Intuitively, $\lambda x{:}\,t.e$ is a characteristic function denoting the set of terms of type t that satisfy e (note that x will usually be bound in e), and ee' means $e' \in e$. In HOL, λ-abstraction forms terms of type $t \to 2$, rather than general function types $t \to t'$. The empty record $()$ is the sole inhabitant of the unit type 1. (e, e') means to form a pair of e and e', and $e.1$ and $e.2$ mean to project the first and second components of e, respectively. $\mathit{ff}_t\ e$ effectively means "e is impossible"; i.e., ff_t eliminates terms of type 0. $\mathit{inl}_t\ e$ and $\mathit{inr}_t\ e$ tag e with the choice of "left" or "right", and the $\mathit{case}\ e\ \mathit{of}\ \ldots$ construct allows conditional execution based on whether the tag of e is "left" or "right". Finally, $e = e'$ tests e and e' for equality.

To make the above intuition precise, we will now define a typing relation between terms and types. A *context* Γ is a list of bindings of variables to types:

$$\Gamma ::= - \mid \Gamma, x{:}t$$

The three-place *typing relation* $\Gamma \vdash e{:}t$ is inductively defined using inference rules [64] as:

$$\text{VAR1}\ \frac{}{\Gamma, x{:}t \vdash x{:}t} \qquad \text{VAR2}\ \frac{\Gamma \vdash x{:}t}{\Gamma, y{:}s \vdash x{:}t} \qquad \text{ABS}\ \frac{\Gamma, x{:}t \vdash e : 2}{\Gamma \vdash \lambda x{:}t.e : t \to 2} \qquad \text{APP}\ \frac{\Gamma \vdash f : t \to 2 \qquad \Gamma \vdash e : t}{\Gamma \vdash fe : 2}$$

$$\text{UNIT}\ \frac{}{\Gamma \vdash () : 1} \qquad \text{VOID}\ \frac{\Gamma \vdash e : 0}{\Gamma \vdash \mathit{ff}_t\ e : t} \qquad \text{PROJ1}\ \frac{\Gamma \vdash e : s \times t}{\Gamma \vdash e.1 : s} \qquad \text{PROJ2}\ \frac{\Gamma \vdash e : s \times t}{\Gamma \vdash e.2 : t}$$

$$\text{PAIR}\ \frac{\Gamma \vdash e : s \qquad \Gamma \vdash f : t}{\Gamma \vdash (e, f) : s \times t} \qquad \text{EQ}\ \frac{\Gamma \vdash e : t \qquad f : t}{\Gamma \vdash e = f : 2}$$

$$\text{INL}\ \frac{\Gamma \vdash e : s}{\Gamma \vdash \mathit{inl}_t\ e : s + t} \qquad \text{INR}\ \frac{\Gamma \vdash e : s}{\Gamma \vdash \mathit{inr}_t\ e : t + s} \qquad \text{CASE}\ \frac{\Gamma \vdash e : s + t \qquad \Gamma, x{:}s \vdash f : u \qquad \Gamma, y{:}t \vdash g : u}{\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g : u}$$

The other operations commonly associated with HOL, such as propositional logic and universal and existential quantifiers, can be defined in terms of the above operations as follows:

$$\top\ :=\ () = () \qquad \bot\ :=\ \forall x{:}2.x \qquad p \wedge q\ :=\ (p, q) = (\top, \top) \qquad p \Rightarrow q\ :=\ p \wedge q = p$$

$$\forall x{:}t.\varphi\ :=\ \lambda x{:}t.\varphi = \lambda x{:}t.\top \qquad \exists x{:}t.\varphi\ :=\ \forall y{:}2.(\forall x{:}t.\varphi \Rightarrow y) \Rightarrow y$$

$$p \vee q\ :=\ \forall x{:}2.((p \Rightarrow x) \wedge (q \Rightarrow x)) \Rightarrow x \qquad \neg p\ :=\ p \Rightarrow \bot$$

3.2.1 Entailment

The *entailment relation* $\varphi \Vdash \psi$ between HOL propositions (terms of type 2) is defined below in the usual way [53]. To allow for empty types such as 0, we give a family of entailment relations $\varphi \Vdash_\Gamma \psi$, each indexed by a typing context for φ, ψ. A sentence σ is *provable* if $\top \Vdash \sigma$, also written $\Vdash \sigma$. A semantics for HOL is *sound* when it equates the meanings of HOL terms that are provably equal according to these rules. In this chapter we will not work directly with HOL's entailment relation, but the existence of the entailment relation is why HOL may be properly called a *logic*.

1. Classic:

   (a) $\top \Vdash \forall p.p \vee \neg p$

2. Order

   (a) $\varphi \Vdash_\Gamma \varphi$

   (b) $\varphi(x) \Vdash_{\Gamma;x} \psi(x)$ implies $\varphi(e) \Vdash_\Gamma \psi(e)$

   (c) $\varphi \Vdash_\Gamma \psi$ and $\psi \Vdash_\Gamma \vartheta$ implies $\varphi \Vdash_\Gamma \vartheta$

3. Equality

   (a) $\top \Vdash_\Gamma e = e$

   (b) $\vartheta \Vdash_\Gamma \varphi \Rightarrow \psi$ and $\vartheta \Vdash_\Gamma \psi \Rightarrow \varphi$ implies $\vartheta \Vdash_\Gamma \varphi = \psi$

   (c) $e = e' \Vdash_{\Gamma;x} \varphi(e) \Rightarrow \varphi(e')$

   (d) $\forall x, fx = f'x \Vdash_\Gamma f = f'$

4. Products

   (a) $\top \Vdash_\Gamma (e_1, e_2).1 = e_1$ (c) $\top \Vdash_\Gamma (e.1, e.2) = e$

   (b) $\top \Vdash_\Gamma (e_1, e_2).2 = e_2$ (d) $\top \Vdash_\Gamma \forall x : 1.x = ()$

5. Co-products

   (a) $\top \Vdash_{\Gamma;x} \mathit{case}\ \mathit{inj}_1\ x\ \mathit{of}\ \lambda x.e_1\ \mathit{or}\ \lambda x.e_2 = e_1$

   (b) $\top \Vdash_{\Gamma;x} \mathit{case}\ \mathit{inj}_2\ x\ \mathit{of}\ \lambda x.e_1\ \mathit{or}\ \lambda x.e_2 = e_2$

   (c) $\top \Vdash_\Gamma \mathit{case}\ f\ \mathit{of}\ \lambda x.\mathit{inj}_1\ x\ \mathit{or}\ \lambda x.\mathit{inj}_2\ x = f$

   (d) $\top \Vdash_\Gamma \forall f, g{:}0 \to A.f = g$

6. Functions

   (a) $\top \Vdash_{\Gamma;x} (\lambda x.e)x = e$ (b) $\top \Vdash_\Gamma \lambda x.fx = f$ (x not free in f)

7. Elementary logic

   (a) $\bot \Vdash_\Gamma \varphi$

   (b) $\varphi \Vdash_\Gamma \top$

   (c) $\exists x.\vartheta \Vdash_\Gamma \varphi$ iff $\vartheta \Vdash_{\Gamma;x} \varphi$

   (d) $\vartheta \Vdash_\Gamma \forall x.\varphi$ iff $\vartheta \Vdash_{\Gamma;x} \varphi$

   (e) $\varphi \Vdash_\Gamma \neg\psi$ iff $\varphi \wedge \psi \Vdash_\Gamma \bot$

   (f) $\vartheta \Vdash_\Gamma \varphi \wedge \psi$ iff $\vartheta \Vdash_\Gamma \varphi$ and $\vartheta \Vdash_\Gamma \psi$

   (g) $\vartheta \wedge \varphi \Vdash_\Gamma \psi$ iff $\vartheta \Vdash_\Gamma \varphi \Rightarrow \psi$

   (h) $\vartheta \vee \varphi \Vdash_\Gamma \psi$ iff $\vartheta \Vdash_\Gamma \psi$ and $\varphi \Vdash_\Gamma \psi$

It is a seminal result that provable equality in the above system corresponds exactly to semantic equality in the topos semantics defined in the next section. In other words, if two terms denote the same morphism in every topos, the terms are provably equal. In this way, higher-order logic is the *internal logic* of topoi. Completeness no longer holds when we restrict our semantics to the category of sets, which is to say, there are many terms that denote the same morphism in the category of sets but that are not provably equal according to the above rules. This failure is a consequence of Godel's incompleteness theorems [53]. Other semantics are complete for the above rules, including so-called Henkin semantics [53].

Remark. Readers well-versed in type-theory may wonder why there are no *commuting conversions* [56] in the above entailment relation. The reason is that commuting conversions are additional equations (often) necessary to decide equivalence of terms *via re-writing*. Since in this chapter we are not interested in deciding equality of HOL terms, we need not include commuting conversions.

3.2.2 Topoi

The categorical semantics of HOL is given by the notion of a topos [53], in the sense that for any topos $\mathcal{T}$, every typing derivation in HOL denotes a morphism in $\mathcal{T}$. A topos is a category with finite products, co-products, exponentials, and a sub-object classifier. In this chapter we will be working with classical HOL, so we will only be concerned with topoi that are boolean. We will now define what it means for $\mathcal{T}$ to be a boolean topos, first by defining notation for naming particular morphisms in $\mathcal{T}$, and then by giving equations between these named morphisms. To typographically distinguish a morphism $f : A \to B$ from a HOL term of type $A \to 2$, we will often write $A : f : B$.

1. (Products) We will write terminal morphisms in $\mathcal{T}$ as $A : \star^A : 1$. A morphism from 1 is called a "point". We will write the projection morphisms in $\mathcal{T}$ as $A \times B : \pi_1^{A,B} : A$ and $A \times B : \pi_2^{A,B} : B$ and the pairing operation as $A : \langle f, g \rangle : B \times C$ for morphisms $A : f : B$ and $A : g : C$ in $\mathcal{T}$. We abbreviate $A \times X : f \times g : B \times Y := \langle \pi_1; f, \pi_2; g \rangle$ for $A : f : B$ and $X : g : Y$ in $\mathcal{T}$.

2. (Co-products) We will write the initial morphism in $\mathcal{T}$ as $0 : \not{f}^A : A$. We will write the injection morphisms in $\mathcal{T}$ as $A : inj_1^{A,B} : A + B$ and $B : inj_2^{A,B} : A + B$. We will write the co-pairing operation as $A + B : \langle f \oplus g \rangle : C$ for morphisms $A : f : C$ and $B : g : C$ in $\mathcal{T}$. We abbreviate $A + X : f + g : B + Y := \langle f; inj_1 \oplus g; inj_2 \rangle$ for $A : f : B$ and $X : g : Y$ in $\mathcal{T}$. We will write the distributive morphisms in $\mathcal{T}$ as $C \times (A + B) : dist^{A,B,C} : (C + A) \times (C + B)$ and $: (C + A) \times (C + B) : undist^{A,B,C} : C \times (A + B)$.

3. (Exponentials) If A and B are objects then we write A^B for the exponential object. We have morphisms $B^A \times A : ev^{A,B} : B$ and $A : \Lambda f : C^B$ for $A \times B : f : C$ in $\mathcal{T}$.

4. (Sub-object classifier) We will write the sub-object classifier of $\mathcal{T}$ as Ω, and will write $1 : \top : \Omega$ for the "true" morphism such that for every monomorphism $j : U \hookrightarrow X$, there exists a classifying morphism $X : \chi_j : \Omega$ such that $j; \chi_j = \star; \top$. A morphism j is a monomorphism when $j; f = j; g$ implies $f = g$. We will write $A \times A : \delta^A : \Omega$ for the morphism δ in $\mathcal{T}$ that classifies the diagonal morphism $\Delta^A := A : \langle id, id \rangle : A \times A$. A sub-object of an object X is a monomorphism $U \hookrightarrow X$.

5. (Boolean) In a boolean topos, for a given object X, the lattice of monomorphisms $A \hookrightarrow X$ is boolean. This implies that $\Omega \cong 1 + 1$.

6. (Well-pointed) Elementary topoi are well-pointed, meaning that they are "extensional": for every $1 : h : A$, we have that $h; f = h; g$ implies $f = g$.

In summary, we have the following language for describing morphisms in a topos $\mathcal{T}$:

$$\frac{}{A : id : A} \qquad \frac{A : f : B \qquad B : g : C}{A : f; g : B} \qquad \frac{}{A : \star : 1} \qquad \frac{}{0 : \mathit{ff} : A} \qquad \frac{}{A \times B : \pi_1 : A} \qquad \frac{}{A \times B : \pi_2 : B}$$

$$\frac{A : f : B \qquad A : g : C}{A : \langle f, g \rangle : B \times C} \qquad \frac{}{A : inj_1 : A + B} \qquad \frac{}{B : inj_2 : A + B} \qquad \frac{B : f : A \qquad C : g : A}{B + C : \langle f \oplus g \rangle : A}$$

$$\frac{}{(A \to B) \times A : ev : B} \qquad \frac{A \times B : f : C}{A : \Lambda f : B \to C} \qquad \frac{j : A \hookrightarrow B}{B : \chi_j : 2} \qquad \frac{}{A \times A : \delta : 2}$$

$$\frac{}{C \times (A + B) : dist : (C + A) \times (C + B)}$$

and a topos $\mathcal{T}$ is such that the following equations hold:

ID-1	ID-2	ASSOC	ETA	BETA	PAIR-BETA-1
$id; f = f$	$f; id = f$	$f; (g; h) = (f; g); h$	$\Lambda ev = id$	$\Lambda h \times id \ ; \ ev = h$	$\langle f, g\rangle; \pi_1 = f$

PAIR-BETA-2	PAIR-ETA	UNIT-ETA	VOID-ETA	SUM-BETA1
$\langle f, g\rangle; \pi_2 = g$	$\langle f; \pi_1, f; \pi_2\rangle = f$	$f = \star \quad (f : A \to 1)$	$f = \not f \quad (f : 0 \to A)$	$inj_1; \langle f \oplus g\rangle = f$

SUM-BETA2	SUM-ETA
$inj_2; \langle f \oplus g\rangle = g$	$\langle inj_1; f \oplus inj_2; f\rangle = f$

The above list of equations is not complete. For example, in a topos, 1 is not initial (i.e., the topos is not degenerate), but this fact is not implied by the above axioms. However, our proofs will make use of mostly the above equations.

3.2.3 Semantics of HOL in a Topos

Let $\mathcal{T}$ be a topos and $\mathbf{D}$ an object of $\mathcal{T}$. To each type t we inductively define a meaning $[\![t]\!]^{\mathcal{T}}_{\mathbf{D}} \in Obj(\mathcal{T})$ as:

$$[\![D]\!] := \mathbf{D} \qquad [\![1]\!] := 1, \text{ the terminal object in } \mathcal{T} \qquad [\![0]\!] := 0, \text{ the initial object in } \mathcal{T}$$

$$[\![s \times t]\!] := [\![s]\!] \times [\![t]\!], \text{ the product in } \mathcal{T} \qquad [\![s + t]\!] := [\![s]\!] + [\![t]\!], \text{ the co-product in } \mathcal{T}$$

$$[\![t \to 2]\!] := [\![2]\!]^{[\![t]\!]}, \text{ the exponential in } \mathcal{T}$$

To each typing derivation $\Gamma \vdash e : t$ we inductively define a meaning as a morphism in $\mathcal{T}$:

$$[\![\Gamma \vdash e : t]\!] \ : [\![\Gamma]\!] \to [\![t]\!]$$

where by $[\![\Gamma]\!]$ we mean the product

$$[\![-]\!] := 1 \qquad [\![\Gamma; x{:}t]\!] := [\![\Gamma]\!] \times [\![t]\!]$$

The exact semantics is [53]:

VAR1-SEM

$[\![\Gamma, x{:}t \vdash x : t]\!] := \pi_2$

VAR2-SEM

$[\![\Gamma, y{:}s \vdash x : t]\!] := \pi_1; [\![\Gamma \vdash x : t]\!]$

UNIT-SEM

$[\![\Gamma \vdash ()]\!] := \star_{[\![\Gamma]\!]}$

PAIR-SEM

$[\![\Gamma \vdash (e, f) : s \times t]\!] := \langle [\![\Gamma \vdash e : s]\!], [\![\Gamma \vdash f : t]\!] \rangle$

PROJ1-SEM

$[\![\Gamma \vdash e.1 : t]\!] := [\![\Gamma \vdash e : s \times t]\!]; \pi_1$

PROJ2-SEM

$[\![\Gamma \vdash e.2 : t]\!] := [\![\Gamma \vdash e : s \times t]\!]; \pi_2$

VOID-SEM

$[\![\Gamma \vdash \mathit{ff}\ e : t]\!] := [\![\Gamma \vdash e : 0]\!]; \mathit{ff}$

INL-SEM

$[\![\Gamma \vdash \mathit{inl}_t\ e : s + t]\!] := [\![\Gamma \vdash e : s]\!]; \mathit{inj}_1$

INR-SEM

$[\![\Gamma \vdash \mathit{inr}_t\ e : t + s]\!] := [\![\Gamma \vdash e : s]\!]; \mathit{inj}_2$

CASE-SEM

$[\![\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x.g\ \mathit{else}\ \lambda y.g]\!] := \langle \mathit{id}, [\![\Gamma \vdash e : s + t]\!] \rangle; \mathit{dist}; \langle [\![\Gamma, x{:}s \vdash f : u]\!] \oplus [\![\Gamma, y{:}t \vdash g : u]\!] \rangle$

ABS-SEM

$[\![\Gamma \vdash \lambda x{:}t.e : t \to 2]\!] := \Lambda [\![\Gamma, x{:}t \vdash e : 2]\!]$

APP-SEM

$[\![\Gamma \vdash fe : 2]\!] := \langle [\![\Gamma \vdash f : t \to 2]\!], [\![\Gamma \vdash e : t]\!] \rangle; ev$

EQ-SEM

$[\![\Gamma \vdash e = f : 2]\!] := \langle [\![\Gamma \vdash e : t]\!], [\![\Gamma \vdash f : t]\!] \rangle; \delta$

The set-theoretic semantics of HOL is obtained by choosing $\mathcal{T}$ to be the topos of sets. In the set-theoretic semantics, a domain $\mathbf{D}$ is a set, the meaning of each type is a set, and the meaning of each morphism is a total function. More specifically, $\times$ denotes cartesian product of sets, $+$ denotes disjoint union of sets, 1 denotes any set with one element, 0 denotes the empty set, and $t \to 2$ denotes the set of all functions from t to 2. The meaning of the terms is then fixed by the meaning of the types; for example, π_1 is the first projection morphism of cartesian product, $\star_t$ is the unique function from t to the one element set, etc.

3.3 Nested Relational Calculus

The NRC [82] is a family of languages, and we will be using the following formulation. The set of NRC *types* is inductively defined as:

$$t ::= D \mid 0 \mid 1 \mid t \times t \mid t + t \mid Pt$$

Intuitively, NRC types are the same as HOL types, but we write Pt rather than $t \to 2$ to indicate sets of t. This is because under our set-theoretic semantics, $t \to 2$ will denote characteristic functions, whereas Pt will represent actual sets. The set of NRC *terms* is inductively defined as:

$$e ::= x \mid () \mid (e, e) \mid e.1 \mid e.2 \mid \mathit{ff}\ e \mid \mathit{inl}_t\ e \mid \mathit{inr}_t\ e \mid \mathit{case}\ e\ \mathit{of}\ \lambda x{:}t.e\ \mathit{or}\ \lambda x{:}t.e \mid e = e$$

$$\mid \mathit{for}\ x{:}t\ \mathit{in}\ e.e \mid \mathit{emp} \mid \mathit{sng}\ e \mid e \cup e \mid \mathit{pow}\ e$$

The only syntactic differences between NRC and HOL are 1) instead of unbounded set comprehension with λ, NRC has bounded comprehension with for; 2) there is no NRC term corresponding to ee, as membership can be derived in NRC (see below); and 3) the NRC has emp, $\cup$, sng, and pow, which can be derived in HOL. The three-place *typing relation* $\Gamma \vdash e : t$ is inductively defined as follows, where we have omitted typing rules that coincide with the HOL rules:

$$\text{EMP}\ \frac{}{\Gamma \vdash \mathit{emp}_t : Pt} \qquad \text{SNG}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathit{sng}\ e : Pt} \qquad \text{POW}\ \frac{\Gamma \vdash e : Pt}{\Gamma \vdash \mathit{pow}\ e : P(Pt)} \qquad \text{UNION}\ \frac{\Gamma \vdash e : Pt \qquad \Gamma \vdash f : Pt}{\Gamma \vdash e \cup f : Pt}$$

$$\text{FOR}\ \frac{\Gamma, x{:}s \vdash e : t \qquad \Gamma \vdash f : Ps}{\Gamma \vdash \mathit{for}\ x{:}t\ \mathit{in}\ f.e : Pt}$$

The intuitive set-theoretic semantics is that $sng\ e$ denotes the singleton set $\{e\}$, emp denotes the empty set $\{\}$, $pow\ e$ denotes the power set of e, and for represents a combination of "map" and "union all":

$$\mathit{for}\ x{:}t\ \mathit{in}\ e.\ f(x) = \bigcup_{x:t \in e} f(x)$$

e.g, when $e = \{1, 2, 3\}$, $\mathit{for}\ x{:}t\ \mathit{in}\ e.\ f(x)$ means $f(1) \cup f(2) \cup f(3)$. Using the set-theoretic semantics, many SQL queries can be written in the NRC; for example, this query projects the first column from a binary relation R:

$$R : P(s \times t) \vdash \mathit{for}\ x{:}t \times t\ \mathit{in}\ R.\ \mathit{sng}\ x.1 : Ps$$

This query constructs the cartesian product of two unary relations R and S:

$$R : Pt, S : Pt \vdash \mathit{for}\ x{:}t\ \mathit{in}\ R.\ \mathit{for}\ y{:}t\ \mathit{in}\ S.\ \mathit{sng}\ (x.1, x.2) : P(s \times t)$$

We will abbreviate:

$$if\ b\ then\ e\ else\ f\ :=\ case\ b\ of\ \lambda x{:}1.e\ or\ \lambda y{:}1.f$$

$$for\ x\ in\ X\ where\ p\ return\ e\ :=\ for\ x\ in\ X.\ if\ P\ then\ sng\ e\ else\ emp$$

$$mem\ x\ X\ :=\ emp_1\ \neq\ for\ (y\ in\ X)\ where\ y = x\ return\ ()$$

$$cartprod\ X\ Y\ :=\ for\ x\ in\ X.for\ y\ in\ Y.\ sng\ (x, y)$$

$$disjunion\ X\ Y\ :=\ for\ x\ in\ X.sng\ inl\ x\ \cup\ for\ y\ in\ Y.sng\ inr\ y$$

3.3.1 The Power Monad

The categorical semantics of the NRC is defined in terms of a monad [82] (see chapter 1 for a definition of monads). Hence, to give a semantics to the NRC in a boolean topos $\mathcal{T}$ we must construct a monad on $\mathcal{T}$. To be faithful to the NRC's set-theoretic semantics, this monad should correspond to the power-set monad when $\mathcal{T}$ is the topos of sets. Fortunately, every topos comes equipped with just such a monad, because every topos is cartesian closed and has a sub-object classifier, Ω. Let $\mathcal{P} : \mathcal{T} \to \mathcal{T}$ the *power monad* taking objects X to exponential objects Ω^X and morphisms $X \to Y$ to morphisms $\Omega^X \to \Omega^Y$. In the topos of sets, the power monad is indeed naturally isomorphic (but not equal) to power-set monad, and when working set-theoretically we will use the power-set monad instead of the power monad. We will write

- $\mathcal{P}A : pow_A : \mathcal{P}(\mathcal{P}A)$ for the power morphism. Set-theoretically, $pow(X) = \{Y \mid Y \subseteq X\}$.
- $A : \eta_A : \mathcal{P}A$ and $\mathcal{P}(\mathcal{P}A) : \mu_A : \mathcal{P}A$ for $\mathcal{P}$'s unit and join natural transformations, respectively. Set-theoretically, η_A maps a set A to itself in $\mathcal{P}(A)$, and μ_A unions a set of sets of A into a set of A.

In fact, the power monad will also

- have a *zero*: a morphism $1 : \emptyset : \mathcal{P}A$. Set-theoretically, *zero* returns the empty set.
- have a *commutative* and *idempotent plus*: a morphism $\mathcal{P}A \times \mathcal{P}A : plus : \mathcal{P}A$. Set theoretically, *plus* is binary union.
- have a strength: a morphism $A \times \mathcal{P}B : \rho_{A,B} : \mathcal{P}(A \times B)$. Set-theoretically,

$$\rho(a, \{b_1, \ldots, b_n\}) = \{(a, b_1), \ldots, (a, b_n)\}$$

In summary, we have extended our language for describing morphisms in a topos $\mathcal{T}$ with:

$$\frac{A : f : B}{\mathcal{P}A : \mathcal{P}\ f : \mathcal{P}B} \qquad \frac{}{1 : \emptyset : A} \qquad \frac{}{\mathcal{P}A \times \mathcal{P}A : plus : \mathcal{P}A} \qquad \frac{}{A \times \mathcal{P}B : \rho : \mathcal{P}(A \times B)} \qquad \frac{}{\mathcal{P}(\mathcal{P}A) : \mu : \mathcal{P}A}$$

$$\frac{}{A : \eta : \mathcal{P}A} \qquad \frac{}{\mathcal{P}A : pow : \mathcal{P}(\mathcal{P}A)}$$

Power monads cannot be axiomatized completely equationally. In practice, the following (incomplete) set of equations are used [82]. Define $\overrightarrow{\to} := \langle \pi_1 \circ \pi_1, \langle \pi_2 \circ \pi_1, \pi_2 \rangle \rangle$:

P-ID	P-COMP	MONAD-1	MONAD-2	MONAD-3	ZERO-1
$\mathcal{P}id = id$	$\mathcal{P}(f;g) = \mathcal{P}f;\mathcal{P}g$	$id = \mu \circ \eta$	$id = \mu \circ \mathcal{P}\eta$	$\mu \circ \mu = \mu \circ \mathcal{P}\mu$	$f;\mathcal{P}(\star;\emptyset);\mu = \emptyset$

ZERO-2	PLUS-ZERO	PLUS-COMM	PLUS-IDEM	STR-1
$\emptyset;\mathcal{P}f;\mu = \emptyset$	$\langle f, \emptyset \rangle; plus = f$	$\langle f, g \rangle; plus = \langle g, f \rangle; plus$	$\langle f, f \rangle; plus = f$	$\mathcal{P}\pi_2 \circ \rho = \pi_2$

STR-2	STR-3	PLUS-ASSOC
$\rho \circ (id \times \eta) = \eta$	$\rho \circ (id \times \mu) = \mu \circ \mathcal{P}\rho \circ \rho$	$\langle \langle f, g \rangle; plus, h \rangle; plus = \langle f, \langle g, h \rangle; plus \rangle; plus$

3.3.2 Semantics of NRC in a topos

Let $\mathcal{T}$ be a topos, $\mathbf{D}$ an object of $\mathcal{T}$, and $\mathcal{P}$ the power monad or a monad naturally isomorphic to the power monad. To each type t we inductively define a meaning $[\![t]\!]^{\mathcal{T}}_{\mathbf{D}} \in Obj(\mathcal{T})$ in the same way as for HOL, but we define:

$$[\![Pt]\!] := \mathcal{P}[\![t]\!], \text{ application of } \mathcal{P}$$

To each typing derivation $\Gamma \vdash e : t$ we inductively define a meaning as a morphism in $\mathcal{C}$:

$$[\![\Gamma \vdash e : t]\!] \ : [\![\Gamma]\!] \to [\![t]\!]$$

The exact semantics is as follows, where we have omitted definitions which are the same as in HOL:

EMP-SEM

$$[\![\Gamma \vdash emp_t]\!] := \star_{[\![\Gamma]\!]};\emptyset_{[\![t]\!]}$$

PLUS-SEM

$$[\![\Gamma \vdash e \cup f : Pt]\!] := \langle [\![\Gamma \vdash e : Pt]\!], [\![\Gamma \vdash f : Pt]\!] \rangle; plus$$

FOR-SEM

$$[\![\Gamma \vdash for\ x{:}t\ in\ e.f : Ps]\!] := \langle id, e \rangle; \rho; \mathcal{P}(f); \mu$$

POW-SEM

$$[\![\Gamma \vdash pow\ e : P(Pt)]\!] := [\![\Gamma \vdash e : Pt]\!]; pow$$

3.4 Translating Types

Having defined the syntax and semantics of HOL and the NRC in the previous two sections, in this section we begin to define our translation [] from HOL to NRC. We translate HOL types to NRC types as:

$$[1] := 1 \qquad [0] := 0 \qquad [D] := D \qquad [s \times t] := [s] \times [t] \qquad [s + t] := [s] + [t] \qquad [t \to 2] := P[t]$$

The above translation can be inverted, yielding a translation $[]^{-1}$ from NRC types to HOL types. Let $\mathcal{T}$ be a topos and $\mathbf{D}$ an object of $\mathcal{T}$.

Lemma (Type Translation is Isomorphism)**.** *For every HOL type t and NRC type t',*

$$[\![t]\!]_{\mathbf{D}} \cong [\![[t]]\!]_{\mathbf{D}} \qquad \textit{and} \qquad [\![t']\!]_{\mathbf{D}} \cong [\![[t']^{-1}]\!]_{\mathbf{D}}$$

Proof. By induction on types, noting that for every object X in a topos $\mathcal{T}$, $\Omega^X \cong \mathcal{P}X$. □

Set-theoretically, the above says that every subset of a set X can be represented by a characteristic function $X \to 2$, and vice versa. To mediate between instances of type t and $[t]$ we need the following auxiliary definitions:

- Denote by $\mathcal{HOL}_{\mathbf{D}}$ the full sub-category of $\mathcal{T}$ where objects are those generated from HOL's types; i.e., objects have the form $[\![t]\!]_{\mathbf{D}}$ for HOL types t.
- Denote by $\mathcal{NRC}_{\mathbf{D}}$ the full sub-category of $\mathcal{T}$ where objects are those generated from NRC's types; i.e., objects have the form $[\![t]\!]_{\mathbf{D}}$ for NRC types t.

In both cases, morphisms are the same as in $\mathcal{T}$, provided their domain and codomain exist in the subcategory. As a consequence, there will be many more morphisms in $\mathcal{NRC}_{\mathbf{D}}$ than can be expressed as NRC terms. We now define two functors, $\downarrow_{\mathbf{D}}\colon \mathcal{HOL}_{\mathbf{D}} \to \mathcal{NRC}_{\mathbf{D}}$ and $\uparrow_{\mathbf{D}}\colon \mathcal{NRC}_{\mathbf{D}} \to \mathcal{HOL}_{\mathbf{D}}$. We will typically omit the subscript $\mathbf{D}$. On objects,

$$\downarrow([\![t]\!]) := [\![[t]]\!]$$

On morphisms,

$$\downarrow(f : [\![s]\!] \to [\![t]\!]) : [\![[s]]\!] \to [\![[t]]\!] := iso_1; f; iso_2$$

where $iso_1 : [\![[s]]\!] \to [\![s]\!]$ and $iso_2 : [\![t]\!] \to [\![[t]]\!]$ are the isomorphism of the above lemma. The inverse functor $\uparrow$ is defined similarly. In effect, we have extended our language for describing morphisms in a topos by

$$\frac{s : f : t}{[s] : \downarrow f : [t]} \qquad \frac{[s] : f : [t]}{s : \uparrow f : t}$$

ISO-1 $\downarrow\uparrow f = f$ ISO-2 $\uparrow\downarrow f = f$

Set-theoretically, $\downarrow$ maps characteristic functions to their specified subset, and $\uparrow$ does the reverse. For example, let $X = \{1, 2\}$. Then

$X \to 2$	$\downarrow$	PX	$\uparrow$	$X \to 2$
$(1 \mapsto \top, 2 \mapsto \top)$		$\{1, 2\}$		$(1 \mapsto \top, 2 \mapsto \top)$
$(1 \mapsto \top, 2 \mapsto \bot)$		$\{1\}$		$(1 \mapsto \top, 2 \mapsto \bot)$
$(1 \mapsto \bot, 2 \mapsto \top)$		$\{2\}$		$(1 \mapsto \bot, 2 \mapsto \top)$
$(1 \mapsto \bot, 2 \mapsto \bot)$		$\{\}$		$(1 \mapsto \bot, 2 \mapsto \bot)$

The effect of $\downarrow$ and $\uparrow$ on functions is similar, for example, the negation morphism $X \to 2 : \neg : X \to 2$ is mapped appropriately:

$$\downarrow \big((1 \mapsto \top, 2 \mapsto \bot) \mapsto (1 \mapsto \bot, 2 \mapsto \top)\big) \;=\; \{1\} \mapsto \{2\}$$

We have, by construction:

Lemma. *$\uparrow_{\mathbf{D}}$ and $\downarrow_{\mathbf{D}}$ are an isomorphism of categories between $\mathcal{HOL}_{\mathbf{D}}$ and $\mathcal{NRC}_{\mathbf{D}}$.*

and hence the following, which are required for our later semantics preservation proof:

Lemma (PRESERVE)**.**

$$\downarrow \star^{[\![t]\!]} = \star^{[\![[t]]\!]} \qquad \downarrow ff^{[\![t]\!]} = f^{[\![[t]]\!]} \qquad \downarrow \pi_1^{[\![s]\!],[\![t]\!]} = \pi_1^{[\![[s]]\!],[\![[t]]\!]} \qquad \downarrow \pi_2^{[\![s]\!],[\![t]\!]} = \pi_2^{[\![[s]]\!],[\![[t]]\!]}$$

$$\downarrow inj_1^{[\![s]\!],[\![t]\!]} = inj_1^{[\![[s]]\!],[\![[t]]\!]} \qquad \downarrow inj_2^{[\![s]\!],[\![t]\!]} = inj_2^{[\![[s]]\!],[\![[t]]\!]} \qquad \downarrow id^{[\![t]\!]} = id^{[\![[t]]\!]} \qquad \downarrow ev^{[\![s]\!],[\![t]\!]} = \downarrow ev^{[\![[s]]\!],[\![[t]]\!]}$$

$$\downarrow \delta^{[\![t]\!]} = \delta^{[\![[t]]\!]} \qquad \downarrow dist^{[\![s]\!],[\![t]\!],[\![u]\!]} = dist^{[\![[s]]\!],[\![[t]]\!],[\![[u]]\!]} \qquad \downarrow \langle f, g\rangle^{[\![s]\!],[\![t]\!],[\![u]\!]} = \langle \downarrow f, \downarrow g\rangle^{[\![[s]]\!],[\![[t]]\!],[\![[u]]\!]}$$

$$\downarrow \langle f \oplus g\rangle^{[\![s]\!],[\![t]\!],[\![u]\!]} = \langle \downarrow f \oplus \downarrow g\rangle^{[\![[s]]\!],[\![[t]]\!],[\![[u]]\!]}$$

Proof. Because $\downarrow$ is an isomorphism of categories. □

3.5 Change of Domain

To proceed further we need to be able to relate the output of an NRC term q under a domain $\mathbf{D_1}$ to the output of q under a different domain $\mathbf{D_2}$. For example, if $[\![q]\!]_{\mathbf{D_1}}$ is the identity function on $\mathbf{D_1}$, then we want to know that $[\![q]\!]_{\mathbf{D_2}}$ is the identity function on $\mathbf{D_2}$.

Let $\mathcal{T}$ be a topos and let $\varphi : \mathbf{D_1} \to \mathbf{D_2}$ a morphism in $\mathcal{T}$. We define by induction on NRC types a morphism $apply_t^{\varphi} : [\![t]\!]_{\mathbf{D_1}} \to [\![t]\!]_{\mathbf{D_2}}$ in $\mathcal{T}$:

$$apply_1 := id \qquad apply_0 := id \qquad apply_{s\times t} := apply_s \times apply_t \qquad apply_{s+t} := apply_s + apply_t$$

$$apply_{Pt} := \mathcal{P}apply_t \qquad\qquad apply_d := \varphi$$

Intuitively, $apply_t^{\varphi}$ transforms instances of type t over domain $\mathbf{D_1}$ to be instances of type t over domain $\mathbf{D_2}$. When φ is an inclusion $apply_t^{\varphi}$ is the identity.

Lemma (PUSH-APPLY)**.**

$$apply; \langle f, g\rangle = \langle apply; f, apply; g\rangle \qquad \langle f, g\rangle; apply = \langle f; apply, g; apply\rangle$$

Proof. Routine - see appendix. □

Lemma (PUSH-↓)**.**

$$apply^{\varphi}_{[\![t]\!]}; (\downarrow f) = \downarrow (apply^{\varphi}_{[\![t]\!]}; f) \qquad (\downarrow f); apply^{\varphi}_{[\![t]\!]} = \downarrow (f; apply^{\varphi}_{[\![t]\!]})$$

Proof. By PRESERVE and that ↓ is a functor. □

3.6 The Active Domain and Universe Queries

In this section we define an NRC expression that computes the active universe of type t on an input instance I of type t'. Intuitively, we first examine t' to "shred" I into a set of constants—its active domain. Then, we examine t to build a new set corresponding to the active universe of type t. More formally, we define, for each Γ, an NRC expression $U_t(\Gamma)$ such that $\Gamma \vdash U_t(\Gamma) : Pt$.

- First, we define by induction on types a function $atoms$ that maps NRC expressions e such that $\Gamma \vdash e : t$ to NRC expressions $atoms(e)$ such that $\Gamma \vdash atoms(e) : PD$. Intuitively, if e denotes a complex object of type t, then $atoms(e)$ denotes all the set of all "atoms", or constants, of type D that are contained in e.

$$atoms_0(e) := emp \qquad atoms_1(e) := emp \qquad atoms_D(e) := sng\ e$$

$$atoms_{s\times t}(e) := atoms_s(e.1) \cup atoms_t(e.2) \qquad atoms_{Pt}(e) := for\ x{:}t\ in\ e.\ atoms_t(x)$$

$$atoms_{s+t}(e) := case\ e\ of\ \lambda x{:}s.atoms_s(x)\ or\ \lambda y{:}t.atoms_t(y)$$

  We extend $atoms$ to map contexts Γ to terms $atoms(\Gamma)$ such that $\Gamma \vdash atoms(\Gamma) : PD$:

$$atoms(-) := atoms(()) \qquad atoms(\Gamma, x{:}t) := atoms(x) \cup atoms(\Gamma)$$

- Second, we define by induction on types a function $univ_t$ that maps NRC expressions e such that $\Gamma \vdash e : PD$ to NRC expressions $univ_t(e)$ such that $\Gamma \vdash univ_t(e) : Pt$. Intuitively, if e denotes an "active domain" of type PD, then $univ_t(e)$ is the "active universe" of type Pt.

$$univ_1(e) := sng\ () \qquad univ_0(e) := sng\ () \qquad univ_D(e) := e \qquad univ_{Pt}(e) := pow\ univ_t(e)$$

$$univ_{s\times t}(e) := cartprod\ univ_s(e)\ univ_t(e) \qquad univ_{s+t}(e) := disunion\ univ_s(e)\ univ_t(e)$$

- Finally, for each Γ, we define an NRC expression $U_t(\Gamma)$ such that $\Gamma \vdash U_t(\Gamma) : Pt$ as $U_t(\Gamma) := univ_t(atoms(\Gamma))$.

Remark. This is the only place where we require the *pow* operation of the NRC. Thus, we can imagine translating from HOL to the NRC extended with U_t instead of *pow*.

3.6.1 Semantics

Having defined the active universe query for NRC we now prove a theorem about its semantics. Let $\mathbf{D}$ be an object in $\mathcal{T}$, and Γ an NRC context. For every point $1 : I : [\![\Gamma]\!]_{\mathbf{D}}$, there exists a distinguished sub-object of $\mathbf{D}$, which we call $adom(I)$. We will write $adom(I)$ for both the object and morphism parts of this

sub-object; i.e., $adom(I) : adom(I) \hookrightarrow \mathbf{D}$. $adom(I)$ is defined as follows. Consider the morphism

$$1 : f : D \to 2 \ := \uparrow (I; [\![\Gamma \vdash atoms(\Gamma) : PD]\!]_{\mathbf{D}})$$

then $1 \times \mathbf{D} : \Lambda f : 2$, and since $1 \times \mathbf{D} \cong \mathbf{D}$, we have a morphism $\mathbf{D} : f' : 2$. In a topos, morphisms from $\mathbf{D}$ to 2 are bijective with sub-objects of $\mathbf{D}$. The main lemma of this section states that the active universe $U_t(I)$ can be computed as the entire universe $\lambda x{:}t.\top$ restricted to the active domain of I:

Lemma (U-SEM)**.** *In the topos of sets, for every point* $1 : I : [\![\Gamma]\!]_{\mathbf{D}}$*,*

$$I; [\![\Gamma \vdash U_{[t]}(\Gamma) : P[t]]\!]_{\mathbf{D}} \ = \downarrow [\![- \vdash \lambda x{:}t.\top : t \to 2]\!]_{adom(I)};\ apply^{P[t]}_{adom(I)}$$

Proof. By induction on t, each side is the meaning of t under the active domain, $[\![[t]]\!]_{adom(I)}$.

□

3.7 The Translation HOL to NRC

We are finally ready to define a translation [] from HOL terms to NRC terms. The translation maps HOL typing derivations $\Gamma \vdash e : t$ to NRC typing derivations $[\Gamma \vdash e : t]$. The translation is homomorphic on all typing rules except for APP and ABS. The key idea is to translate λ abstraction as for abstraction over the active universe.

ABS-TRANS

$$\frac{[\Gamma, x{:}t \vdash e : 2] = [\Gamma], x{:}[t] \vdash e' : 2}{[\Gamma \vdash \lambda x{:}t.e : t \to 2] := [\Gamma] \vdash for\ x{:}[t]\ in\ U([\Gamma]).e' : Pt}$$

APP-TRANS

$$\frac{[\Gamma \vdash f : t \to 2] = [\Gamma] \vdash f' : P[t] \qquad [\Gamma \vdash e : t] = [\Gamma] \vdash e' : [t]}{[\Gamma \vdash fe : 2] := [\Gamma] \vdash e'\ mem\ f' : 2}$$

VAR1-TRANS

$$\frac{}{[\Gamma, x{:}t \vdash x{:}t] := [\Gamma], x{:}[t] \vdash x{:}[t]}$$

VAR2-TRANS

$$\frac{[\Gamma \vdash x{:}t] = [\Gamma] \vdash x{:}[t]}{[\Gamma, y{:}s \vdash x{:}t] := [\Gamma], y : [s] \vdash x{:}[t]}$$

EQ-TRANS

$$\frac{[\Gamma \vdash e : t] = [\Gamma] \vdash e' : [t] \qquad [\Gamma \vdash f : t] = [\Gamma] \vdash f' : [t]}{[\Gamma \vdash e = f : 2] := [\Gamma] \vdash e' = f' : 2}$$

CASE-TRANS

$$\frac{[\Gamma \vdash e : s + t] = [\Gamma] \vdash e' : [s] + [t] \qquad [\Gamma, x{:}s \vdash f : u] = [\Gamma], x{:}[s] \vdash f' : [u] \qquad [\Gamma, y{:}t \vdash g : u] = [\Gamma], y{:}[t] \vdash g' : [u]}{[\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g\ : u] := [\Gamma] \vdash \mathit{case}\ e'\ \mathit{of}\ \lambda x{:}[s].f'\ \mathit{or}\ \lambda y{:}[t].g' : [u]}$$

UNIT-TRANS

$$\frac{}{[\Gamma \vdash ()] := [\Gamma] \vdash () : 1}$$

VOID-TRANS

$$\frac{[\Gamma \vdash e : 0] = [\Gamma] \vdash e' : [0]}{[\Gamma \vdash \mathit{ff}\ e : t] := [\Gamma] \vdash \mathit{ff}\ e' : [t]}$$

PROJ1-TRANS

$$\frac{[\Gamma \vdash e : t \times s] = [\Gamma] \vdash e' : [t] \times [s]}{[\Gamma \vdash e.1 : t] := [\Gamma] \vdash e'.1 : [t]}$$

PROJ2-TRANS

$$\frac{[\Gamma \vdash e : t \times s] = [\Gamma] \vdash e' : [t] \times [s]}{[\Gamma \vdash e.2 : t] := [\Gamma] \vdash e'.2 : [s]}$$

PAIR-TRANS

$$\frac{[\Gamma \vdash e : s] = [\Gamma] \vdash e' : [s] \qquad [\Gamma \vdash f : t] = [\Gamma] \vdash f' : [t]}{[\Gamma \vdash (e, f) : s \times t] := [\Gamma] \vdash (e', f') : [s] \times [t]}$$

INL-TRANS

$$\frac{[\Gamma \vdash e : s] = [\Gamma] \vdash e' : [s]}{[\Gamma \vdash \mathit{inl}_t\ e : s + t] := [\Gamma] \vdash \mathit{inl}_{[t]}\ e' : [s] + [t]}$$

INL-TRANS

$$\frac{[\Gamma \vdash e : s] = [\Gamma] \vdash e' : [s]}{[\Gamma \vdash \mathit{inr}_t\ e : t + s] := [\Gamma] \vdash \mathit{inr}_{[t]}\ e' : [t] + [s]}$$

3.8 Domain independence

Because the translation from HOL to NRC translates unbounded quantification to bounded quantification over the active universe, it will only be semantics preserving on HOL typing derivations that are *domain-independent*. The traditional notion of domain independence for complex objects [78] generalizes to our categorical setting as follows. Let s and t be NRC types. An object-indexed family of morphisms in $\mathcal{T}$:

$$q_D : [\![s]\!]_D \ \to [\![t]\!]_D$$

is *domain-independent* when for every monomorphism $\varphi : \mathbf{D_1} \hookrightarrow \mathbf{D_2}$ in $\mathcal{T}$:

$$\mathit{apply}_s^{\varphi}; q_{\mathbf{D_2}} = q_{\mathbf{D_1}}; \mathit{apply}_t^{\varphi} \qquad (DI - SEM)$$

If we write $I_1 \sqsubseteq_{\varphi}^{t} I_2$ to mean $\mathit{apply}_{\varphi}^{t} \circ I_1 = I_2$, then the above condition can be rendered in "logical relations" form [64]:

$$I_1 \sqsubseteq_f^s I_2 \quad \text{implies} \quad q_{D_1} \circ I_1 \sqsubseteq_f^t q_{D_2} \circ I_2$$

i.e., domain-independent families of morphisms map $\sqsubseteq$-related inputs to $\sqsubseteq$-related outputs. When φ is an inclusion function, $apply^{\varphi}$ is the identity and the above becomes

$$I_1 = I_2 \quad \text{implies} \quad q_{D_1} \circ I_1 = q_{D_2} \circ I_2$$

which corresponds exactly to the first-order notion of domain independence discussed in the introduction. Domain independence is also captured by a commutative diagram:

$$\begin{array}{ccc} \mathbf{D_1} & [\![s]\!]_{\mathbf{D_1}} & \xrightarrow{q_{\mathbf{D_1}}} [\![t]\!]_{\mathbf{D_1}} \\ {\scriptstyle\varphi}\downarrow & {\scriptstyle apply_s^{\varphi}}\downarrow & \downarrow{\scriptstyle apply_t^{\varphi}} \\ \mathbf{D_2} & [\![s]\!]_{\mathbf{D_2}} & \xrightarrow[q_{\mathbf{D_2}}]{} [\![t]\!]_{\mathbf{D_2}} \end{array}$$

Let $\mathcal{T}_{\hookrightarrow}$ denote the sub-category of $\mathcal{T}$ such that the morphisms of $\mathcal{T}_{\hookrightarrow}$ are the monomorphisms of $\mathcal{T}$. If we think of $[\![t]\!]_-$, for each t, as a functor from $\mathcal{T}_{\hookrightarrow}$ to $\mathcal{T}$ (where the action of $[\![t]\!]_-$ on morphisms is given by $apply_t^-$), then our notion of domain independence means that our family of morphisms q is a natural transformation from the functor $[\![s]\!]_-$ to the functor $[\![t]\!]_-$:

$$\mathcal{T}_{\hookrightarrow} \;\underset{[\![t]\!]_-}{\overset{[\![s]\!]_-}{\Downarrow q_-}}\; \mathcal{T}$$

Lemma (DI-COMP)**.** *If* $[\![A]\!]_{\mathbf{D}} : f_{\mathbf{D}} : [\![B]\!]_{\mathbf{D}}$ *and* $[\![B]\!]_{\mathbf{D}} : g_{\mathbf{D}} : [\![C]\!]_{\mathbf{D}}$ *are domain-independent, then so is* $[\![A]\!]_{\mathbf{D}} : f_{\mathbf{D}}; g_{\mathbf{D}} : [\![C]\!]_{\mathbf{D}}$.

Proof. We know that (1) $apply_A; f_{\mathbf{D_1}} = f_{\mathbf{D_2}}; apply_B$ and (2) $apply_B; g_{\mathbf{D_1}} = g_{\mathbf{D_2}}; apply_C$. We must show that $apply_A; f_{\mathbf{D_1}}; g_{\mathbf{D_1}} = f_{\mathbf{D_2}}; g_{\mathbf{D_2}}; apply_C$. Rewriting the goal by (1) and (2) yields $f_{\mathbf{D_2}}; apply_B; g_{\mathbf{D_1}} = f_{\mathbf{D_2}}; apply_B; g_{\mathbf{D_1}}$. □

Example

Define two HOL expressions p and q:

$$p \;:=\; - \vdash \lambda x{:}D.\top : D \to 2 \qquad q \;:=\; - \vdash \lambda x{:}D.\bot : D \to 2$$

That is, p forms the "universal set" of type D, and q forms the "empty set" of type D. As we now show, p is not domain-independent, but q is. Assume we are working in the category of sets. Let $\mathbf{D_1} = \{1\}$, $\mathbf{D_2} = \{1, 2\}$, and $\varphi : \mathbf{D_1} \hookrightarrow \mathbf{D_2}$ be the inclusion function $\varphi(1) = 1$. Since φ is an inclusion, $apply^{\varphi}$ is the identity and hence for q to be domain-independent we require that $q_{\mathbf{D_1}} = q_{\mathbf{D_2}}$ and indeed this is the case, since both sides are equal to the map $- \mapsto \emptyset$. But for p to be domain-independent we would expect that $p_{\mathbf{D_1}} = p_{\mathbf{D_2}}$ which does not hold, as $p_{\mathbf{D_1}}$ is the map $- \mapsto \{\emptyset, \{1\}\}$ and $p_{\mathbf{D_2}}$ is the map $- \mapsto \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

3.8.1 Hereditary Domain Independence

A HOL typing derivation is *hereditarily domain-independent* when it is domain-independent and all of its sub-derivations are hereditarily domain-independent. An example domain-independent derivation that is not hereditarily so is

$$- \vdash (\lambda x{:}D.\top, \lambda x{:}D.\bot).2$$

We conjecture that domain-independent terms always β-normalize into hereditarily domain-independent terms, but have been unable to prove this.

3.9 Semantics Preservation

We are now in a position to prove that our translation from HOL to NRC preserves semantics:

Theorem. *Let $\mathcal{T}$ be the topos of sets and suppose $\Gamma \vdash e : t$ is a hereditarily domain-independent HOL typing derivation. Then, for every object $\mathbf{D}$ of $\mathcal{T}$,*

$$\downarrow_{\mathbf{D}} [\![\Gamma \vdash e : t]\!]_{\mathbf{D}} \;=\; [\![[\Gamma \vdash e : t]]\!]_{\mathbf{D}}$$

Proof. By induction on the typing derivation $\Gamma \vdash e : t$. The routine cases are in the appendix, and the heart of the proof is the translation of ABS. We will omit the subscript $\mathbf{D}$ and superscript $\mathcal{T}$ wherever possible.

Suppose $\Gamma \vdash e : t$ is of the form

$$\frac{\Gamma, x{:}t \vdash e : 2}{\Gamma \vdash \lambda x{:}t.e : t \to 2} \qquad \text{ABS-SEM} \quad \frac{\Gamma, x{:}t \vdash e : 2}{[\![\Gamma \vdash \lambda x{:}t.e : t \to 2]\!] := \Lambda[\![\Gamma, x{:}t \vdash e : 2]\!]}$$

$$\text{ABS-TRANS} \quad \frac{[\Gamma, x{:}t \vdash e : 2] = [\Gamma], x{:}[t] \vdash e' : 2}{[\Gamma \vdash \lambda x{:}t.e : t \to 2] := [\Gamma] \vdash for\ x{:}[t]\ in\ U([\Gamma]).if\ e'\ then\ sng\ x\ else\ emp : Pt}$$

Our inductive hypothesis is

$$di([\![\Gamma, x{:}t \vdash e : 2]\!]) \text{ implies } \downarrow [\![\Gamma, x{:}t \vdash e : 2]\!] \ = \ [\![[\Gamma, x{:}t \vdash e : 2]]\!]$$

We wish to show that

$$\downarrow [\![\Gamma \vdash \lambda x{:}t.e : t \to 2]\!] \ = \ [\![[\Gamma \vdash \lambda x{:}t.e : t \to 2]]\!]$$

We calculate (where we will prove step BOUND momentarily):

$$\begin{aligned}
&\downarrow [\![\Gamma \vdash \lambda x{:}t.e : t \to 2]\!] \\
&\qquad = \quad ABS-SEM \\
&\downarrow \Lambda[\![\Gamma, x{:}t \vdash e : 2]\!] \\
&\qquad = \quad ISO \\
&\downarrow \Lambda \uparrow\downarrow [\![\Gamma, x{:}t \vdash e : 2]\!] \\
&\qquad = \quad IH \\
&\downarrow \Lambda \uparrow [\![[\Gamma, x{:}t \vdash e : 2]]\!] \\
&\qquad = \quad ABS-TRANS \\
&\downarrow \Lambda \uparrow [\![[\Gamma], x{:}[t] \vdash e' : 2]\!] \\
&\qquad = \quad BOUND \\
&[\![[\Gamma] \vdash for\ x{:}[t]\ in\ U([\Gamma]).if\ e'\ then\ sng\ x\ else\ emp : Pt]\!] \\
&\qquad = \quad ABS-TRANS \\
&[\![[\Gamma \vdash \lambda x{:}t.e : t \to 2]]\!]
\end{aligned}$$

We prove the step $BOUND$ as follows. We want to show that

$$\downarrow \Lambda \uparrow [\![[\Gamma], x\!:\![t] \vdash e' : 2]\!]$$

$$=$$

$$[\![[\Gamma] \vdash for\ x\!:\![t]\ in\ U([\Gamma]).if\ e'\ then\ sng\ x\ else\ emp : P[t]]\!]$$

We will refer to the upper and low parts of the above equation as lhs and rhs, respectively. Let $\mathbf{D}$ be a set. Then lhs and rhs denote functions:

$$[\![[\Gamma]]\!]_{\mathbf{D}} \to \mathcal{P}[\![[t]]\!]_{\mathbf{D}}$$

Let $I \in [\![[\Gamma]]\!]_{\mathbf{D}}$ be a set. Then $lhs(I)$ and $rhs(I)$ are both sets of $[\![[t]]\!]_{\mathbf{D}}$s. In particular, we know that $rhs(I) \subseteq lhs(I)$:

$$\big(\downarrow \Lambda \uparrow [\![[\Gamma], x\!:\![t] \vdash e' : 2]\!]_{\mathbf{D}}\big)(I) \bigcap [\![[\Gamma \vdash U([\Gamma])]\!]_{\mathbf{D}}(I)$$

$$=$$

$$[\![[\Gamma] \vdash for\ x\!:\![t]\ in\ U([\Gamma]).if\ e'\ then\ sng\ x\ else\ emp : P[t]]\!]_{\mathbf{D}}(I)$$

So, to prove $BOUND$ it suffices to show:

$$\big(\downarrow \Lambda \uparrow [\![[\Gamma], x\!:\![t] \vdash e' : 2]\!]_{\mathbf{D}}\big)(I) \ \subseteq\ [\![[\Gamma \vdash U([\Gamma])]\!]_{\mathbf{D}}(I) \tag{3.1}$$

If it were the case that lhs was an arbitrary function, then when applied to I lhs may yield an arbitrary set of $[\![[t]]\!]_{\mathbf{D}}$s, and (3.1) will be false. However, we know that lhs is not an arbitrary function, but is domain-independent—by above, it is equal to $[\![\Gamma \vdash \lambda x\!:\!t.e]\!]$, which we assumed was domain-independent. This means that for any sets $\mathbf{D_1}$ and $\mathbf{D_2}$, any injective function $f : \mathbf{D_1} \hookrightarrow \mathbf{D_2}$, every $I \in [\![\Gamma]\!]_{\mathbf{D_1}}$, and every $J \in [\![\Gamma]\!]_{\mathbf{D_2}}$,

$$I \sqsubseteq_f^{[\Gamma]} J \text{ implies } \big(\downarrow \Lambda \uparrow [\![[\Gamma], x\!:\![t] \vdash e' : 2]\!]\big)_{\mathbf{D_1}}(I) \ \sqsubseteq_f^{[t]}\ \big(\downarrow \Lambda \uparrow [\![[\Gamma], x\!:\![t] \vdash e' : 2]\!]\big)_{\mathbf{D_2}}(J)$$

We choose $\mathbf{D_1}$ to be $atoms(I)$ and $\mathbf{D_2}$ to be $\mathbf{D}$, so that $atoms(I) \subseteq \mathbf{D}$ and $f(x) = x$ is the obvious set inclusion function. We know that $I \in [\![\Gamma]\!]_{atoms(I)}$ and $I \in [\![\Gamma]\!]_{\mathbf{D}}$, so we can use our I as both I and J.

Since we have an inclusion function rather than just an injection, we can replace $\sqsubseteq$ with $=$:

$$\left(\downarrow \Lambda \uparrow [\![[\Gamma], x{:}[t] \vdash e' : 2]\!]\right)_{atoms(I)}(I) \;=\; \left(\downarrow \Lambda \uparrow [\![[\Gamma], x{:}[t] \vdash e' : 2]\!]\right)_{\mathbf{D}}(I) \tag{3.2}$$

Substituting (3.2) into (3.1) gives a new goal

$$\left(\downarrow \Lambda \uparrow [\![[\Gamma], x{:}[t] \vdash e' : 2]\!]\right)_{atoms(I)}(I) \;\subseteq\; [\![[\Gamma] \vdash U([\Gamma])]\!]_{\mathbf{D}}(I) \tag{3.3}$$

By definition of U (lemma U-SEM), we have that

$$\left(\downarrow \Lambda \uparrow [\![-, x{:}[t] \vdash \top : 2]\!]\right)_{atoms(I)} = [\![[\Gamma] \vdash U([\Gamma])]\!]_{\mathbf{D}}(I) \tag{3.4}$$

Substituting (3.4) into (3.3) gives a new goal

$$\left(\downarrow \Lambda \uparrow [\![[\Gamma], x{:}[t] \vdash e' : 2]\!]\right)_{atoms(I)}(I) \;\subseteq\; \left(\downarrow \Lambda \uparrow [\![-, x{:}[t] \vdash \top : 2]\!]\right)_{atoms(I)} \tag{3.5}$$

Which completes the proof (since $e' \leq \top$). □

3.10 Translating NRC to HOL

The reverse translation $[]^{-1}$, where below we suppress the superscript -1, is

EMP-TRANS

$$\frac{}{[\Gamma \vdash emp_t : Pt] := [\Gamma] \vdash \lambda x{:}[t].\bot : [t] \to 2}$$

SNG-TRANS

$$\frac{[\Gamma \vdash e : t] = [\Gamma] \vdash e' : [t]}{[\Gamma \vdash sng\ e : Pt] := [\Gamma] \vdash \lambda x{:}[t].x = e' : [t] \to 2}$$

POW-TRANS

$$\frac{[\Gamma \vdash e : Pt] = [\Gamma] \vdash e' : [t] \to 2}{[\Gamma \vdash pow\ e : P(Pt)] := [\Gamma] \vdash \lambda x{:}[t] \to 2.\forall y{:}[t].xy \Rightarrow e'y : ([t] \to 2) \to 2}$$

UNION-TRANS

$$\frac{[\Gamma \vdash e : Pt] = [\Gamma] \vdash e' : [t] \to 2 \qquad [\Gamma \vdash f : Pt] = [\Gamma] \vdash f' : [t] \to 2}{[\Gamma \vdash e \cup f \vdash Pt] := [\Gamma] \vdash \lambda x{:}[t].e'x \vee f'x : [t] \to 2}$$

FOR-TRANS

$$\frac{[\Gamma \vdash e : Pt] = [\Gamma] \vdash e' : [t] \to 2 \qquad [\Gamma, x{:}t \vdash f(x) : Ps] = [\Gamma], x{:}[t] \vdash f'(x) : [s] \to 2}{[\Gamma \vdash for\ x{:}t\ in\ e.f : Ps] := [\Gamma] \vdash \lambda y : [s].\exists x{:}[t].e'x \wedge f'(x)y : [s] \to 2}$$

In the foregoing, we have omitted the homomorphic translations which are the same as the translation from HOL to NRC.

Lemma. *Suppose $\Gamma \vdash e : t$ is an NRC typing derivation. Let $\mathcal{T}$ be a topos. Then for every $\mathbf{D} \in Obj(\mathcal{T})$,*

$$\uparrow [\![\Gamma \vdash e : t]\!]_{\mathbf{D}} = [\![[\Gamma \vdash e : t]^{-1}]\!]_{\mathbf{D}}$$

Proof. Routine induction - done in Coq. □

3.11 Related Work

Two pieces of especially related work are a translation of first-order set theory into NRA [1], and a translation of the NRC into SQL [23].

Abiteboul and Beeri define a first-order, many sorted calculus which we call first-order set theory (FOST) [1]:

$$t ::= 1 \mid t \times t \mid Pt \mid D$$

$$e ::= x \mid () \mid (e, e) \mid e.1 \mid e.2 \mid \top \mid \bot \mid e \wedge e \mid e \vee e \mid e \Rightarrow e \mid \neg e \mid e = e \mid \forall x{:}t.e \mid \exists x{:}t.e \mid e \in e \mid e \subseteq e$$

They also define a notion of domain independence for FOST (under set-theoretic, not topos, semantics) and provide a translation of FOST into NRA. In fact, their active domain query is the same as ours. Compared to HOL, FOST lacks λ-abstraction and function application and includes predicates $\in$ and $\subseteq$. An alternative approach to translating from HOL to NRA would be to translate from HOL to FOST by eliminating λ-abstraction in favor of $\forall$ and $\exists$. Our approach in this chapter is the opposite, elimination of $\forall$ and $\exists$ in favor of λ.

The Links language allows certain fragments of HOL to be implemented as a combination of the simply-typed λ-calculus and SQL [23]. Rather than attempt to compute the active domain and perform comprehensions over it as we do, which can be inefficient, Links uses a type and effect system to identify HOL fragments that can be effectively implemented as SQL.

3.12 Future Work

HOL is traditionally understood to restrict function types to the form $t \to 2$ [53]. However, it is easy to extend HOL, the NRC, and our translation to arbitrary function types $t \to s$. Semantically, a topos is cartesian closed for all types, not just 2, so the meaning of such terms is straightforward [53]. We can extend our translation HOL to NRC as follows, by "reifying" functions as relations:

$$[s \to t] := P([s] \times [t]) \quad (t \neq 2)$$

$$\frac{[\Gamma \vdash f : s \to t] = [\Gamma] \vdash f' : P([s] \times [t]) \qquad [\Gamma \vdash e : s] = [\Gamma] \vdash e' : [s]}{[\Gamma \vdash fe : t] := [\Gamma] \vdash \iota\ \big(for\ x\colon s \times t\ in\ f'.where\ x.1 = e'\ return\ x.2\big) : [t]}$$

$$\frac{[\Gamma; x\colon t \vdash e(x) : s] = [\Gamma], x\colon [t] \vdash e'(x) : [s]}{[\Gamma \vdash \lambda x\colon t.e(x) : t \to s] := for\ y\colon t \times s\ in\ U_{[s]\to[t]}(\Gamma).\ where\ y.2 = e'(y.1)\ return\ y : P([t] \times [s])}$$

In the foregoing, we have added a *description operator* $\iota_t : Pt \to t$ to the NRC. Semantically, $\iota\ e$ should choose an element from a non-empty set e. If e is empty, then ι should return an arbitrary value of type t. Hence, this translation may be problematic with empty types such as 0. The description operator ι is often assumed in proof assistants for higher-order logic, such as Isabelle/HOL [66]. Of course, we must also take care that when we generate the active universe $U(\Gamma)$ for a function type $s \to t$, we generate only functional relations $P(s \times t)$, not arbitrary relations. Another subtle point is that we should still translate types $t \to 2$ as Pt, because if we reify $t \to 2$ as $P(t \times 2)$ then translated instances of type $t \to 2$ will in fact contain every constant of type t (since $t \to 2$ must be total).

3.13 Coq Mechanization

We have mechanized in Coq [11] all the results of this chapter, except for one part of the proof of the ABS case of the HOL to NRC translation that requires specializing to the topos of sets and hence can not be done using our particular encoding of HOL in Coq. The entire development is several thousand lines and proceeds similarly to the paper development, except that we use the variable-free categorical syntax for HOL. We include some key definitions here. To model a boolean topos, we assume Coq axioms for classical logic, and equate `Prop` with `bool` using the standard library axiom `excluded_middle_informative`.

```
Inductive ty : Set :=
 | one : ty
 | prop : ty
 | prod : ty -> ty -> ty
 | pow : ty -> ty
 | dom : ty
 | zero : ty
 | sum : ty -> ty -> ty.

Fixpoint inst (t: ty) : Set -> Type :=
 fun (D: Set) =>
 match t with
   | one => unit
   | prod t1 t2 => inst t1 D * inst t2 D
   | pow t' => inst t' D -> Prop
   | dom => D
   | prop => Prop
   | sum t1 t2 => inst t1 D + inst t2 D
   | zero => void
 end.

Fixpoint apply {D1 D2 : Set} (f: D1 -> D2) {t} : inst t D1 -> inst t D2 :=
 match t as t return inst t D1 -> inst t D2 with
     | one => fun I => I
     | prod t1 t2 => fun I =>  (apply f (fst I), apply f (snd I))
     | dom => f
     | pow t' => fun I => Im I (apply f)
     | prop => fun I => I
     | zero => fun I => I
```

```
    | sum t1 t2 => fun I => match I with
                                | inl I' => inl (apply f I')
                                | inr I' => inr (apply f I')
                             end
 end.

Definition di {G t} (e: forall D, inst G D -> inst t D)
 := forall (D1 D2: Set) (f: D1 -> D2) (pf: mono f) I,
       apply f (e D1 I) = e D2 (apply f I) .
    (* hdi is simple, so omit it here *)

Inductive exp : ty -> ty -> Set :=
 | equal : forall {t}, exp (prod t t) prop
 | inj1 : forall {t1 t2}, exp t1 (sum t1 t2)
 | inj2 : forall {t1 t2}, exp t2 (sum t1 t2)
 | case : forall {t1 t2 t3}, exp t1 t3 -> exp t2 t3 -> exp (sum t1 t2) t3
 | id : forall {t}, exp t t
 | comp : forall {t1 t2 t3}, exp t1 t2 -> exp t2 t3 -> exp t1 t3
 | star : forall {t}, exp t one
 | pi1 : forall {t1 t2}, exp (prod t1 t2) t1
 | pi2 : forall {t1 t2}, exp (prod t1 t2) t2
 | pair : forall {t1 t2 t3}, exp t1 t2 -> exp t1 t3 -> exp t1 (prod t2 t3)
 | ev : forall {t}, exp (prod (pow t) t) prop
 | curry : forall {t1 t2}, exp (prod t1 t2) prop -> exp t1 (pow t2)
 | contra : forall {t}, exp zero t
 | boolean1 : exp prop (sum one one)
 | boolean2 : exp (sum one one) prop
 | dist1 : forall {a b c}, exp (prod a (sum b c)) (sum (prod a b) (prod a c))
 | dist2 : forall {a b c}, exp (sum (prod a b) (prod a c)) (prod a (sum b c)).
```

```
Fixpoint denote {G t: ty} (e: exp G t) D : inst G D -> inst t D :=
    match e in exp G t return inst G D -> inst t D with
      | id t => fun I => I
      | comp t1 t2 t3 f g => fun I => denote g D (denote f D I)
      | star t => fun I => tt
      | pi1 t1 t2 => fun I => fst I
      | pi2 t1 t2 => fun I => snd I
      | pair t1 t2 t3 f g => fun I =>  (denote f D I, denote g D I)
      | ev t => fun I => (fst I) (snd I)
      | curry t1 t2 f => fun I =>
                (fun J => denote f D (I, J))
      | equal t => fun I => fst I = snd I
      | inj1 t1 t2 => inl
      | inj2 t1 t2 => inr
      | case t1 t2 t3 f g => fun I => match I with
                                          | inl i => denote f D i
                                          | inr i => denote g D i
                                      end
      | contra t => fun I => match I with end
      | boolean1 => fun I => match excluded_middle_informative I with
                               | left _ => inl tt
                               | right _ => inr tt
                             end
      | boolean2 => fun I => match I with
                               | inl _ => True
                               | inr _ => False
                             end
      | dist1 a b c => fun I => match snd I with
                                  | inl l => inl (fst I, l)
                                  | inr r => inr (fst I, r)
                                end
```

```
      | dist2 a b c => fun I => match I with
                                    | inl l => (fst l, inl (snd l))
                                    | inr r => (fst r, inr (snd r))
                                end
    end.


Inductive expB : ty -> ty -> Type :=
 | equalB : forall {t}, expB (prod t t) prop
 | inj1B : forall {t1 t2}, expB t1 (sum t1 t2)
 | inj2B : forall {t1 t2}, expB t2 (sum t1 t2)
 | caseB : forall {t1 t2 t3}, expB t1 t3 -> expB t2 t3 -> expB (sum t1 t2) t3
 | idB : forall {t}, expB t t
 | compB : forall {t1 t2 t3}, expB t1 t2 -> expB t2 t3 -> expB t1 t3
 | starB : forall {t}, expB t one
 | pi1B : forall {t1 t2}, expB (prod t1 t2) t1
 | pi2B : forall {t1 t2}, expB (prod t1 t2) t2
 | pairB : forall {t1 t2 t3}, expB t1 t2 -> expB t1 t3 -> expB t1 (prod t2 t3)
 | mzero : forall {t}, expB one (pow t)
 | mplus : forall {t}, expB (prod (pow t) (pow t)) (pow t)
 | mjoin : forall {t}, expB (pow (pow t)) (pow t)
 | munit : forall {t}, expB t (pow t)
 | mpow : forall {t}, expB (pow t) (pow (pow t))
 | mmap : forall {t1 t2}, expB t1 t2 -> expB (pow t1) (pow t2)
 | boolean1B : expB prop (sum one one)
 | boolean2B : expB (sum one one) prop
 | str : forall {t1 t2}, expB (prod t1 (pow t2)) (pow (prod t1 t2))
 | contraB : forall {t}, expB zero t
 | dist1B : forall {a b c}, expB (prod a (sum b c)) (sum (prod a b) (prod a c))
 | dist2B : forall {a b c}, expB (sum (prod a b) (prod a c)) (prod a (sum b c)).
```

```
Fixpoint denoteB {G t: ty} (e: expB G t) D : inst G D -> inst t D :=
   match e in expB G t return inst G D -> inst t D with
     | idB t => fun I => I
     | compB t1 t2 t3 f g => fun I => denoteB g D (denoteB f D I)
     | starB t => fun I => tt
     | pi1B t1 t2 => fun I => fst I
     | pi2B t1 t2 => fun I => snd I
     | pairB t1 t2 t3 f g => fun I =>  (denoteB f D I, denoteB g D I)
     | equalB t => fun I => fst I = snd I
     | inj1B t1 t2 => inl
     | inj2B t1 t2 => inr
     | caseB t1 t2 t3 f g => fun I => match I with
                                        | inl i => denoteB f D i
                                        | inr i => denoteB g D i
                                      end
     | mzero t => fun I => Empty_set _
     | mplus t => fun I => Union (fst I) (snd I)
     | mjoin t => fun I => join I
     | munit t => fun I => Singleton I
     | mpow t => fun I => Power_set I
     | mmap t1 t2 f => fun I => Im I (denoteB f D)
     | boolean1B => fun I => match excluded_middle_informative I with
                               | left _ => inl tt
                               | right _ => inr tt
                             end
     | boolean2B => fun I => match I with
                               | inl _ => True
                               | inr _ => False
                             end
     | str t1 t2 => fun I => Im (snd I) (fun J => (fst I, J))
     | contraB t => fun I => match I with end
```

```
      | dist1B a b c => fun I => match snd I with
                                 | inl l => inl (fst I, l)
                                 | inr r => inr (fst I, r)
                               end
      | dist2B a b c => fun I => match I with
                                    | inl l => (fst l, inl (snd l))
                                    | inr r => (fst r, inr (snd r))
                               end
    end.

Fixpoint holToNrc {G t: ty} (e: exp G t) : expB G t :=
    match e in exp G t return expB G t with
      | id t => idB
      | comp t1 t2 t3 f g => compB (holToNrc f) (holToNrc g)
      | star t => starB
      | pi1 t1 t2 => pi1B
      | pi2 t1 t2 => pi2B
      | pair t1 t2 t3 f g => pairB (holToNrc f) (holToNrc g)

      | ev t =>
        let rhs := compB starB munit in
        let xxx := whereB (compB (pairB pi2B (compB pi1B pi2B)) equalB)
                          (compB starB munit) pi1B in
        compB (pairB xxx rhs) equalB
      | curry t1 t2 f => whereB (holToNrc f) (compB pi2B munit) UB
      | equal t => equalB
      | inj1 t1 t2 => inj1B
      | inj2 t1 t2 => inj2B
      | case t1 t2 t3 f g => caseB (holToNrc f) (holToNrc g)
      | contra t => contraB
      | boolean1 => boolean1B
```

```
      | boolean2 => boolean2B
      | dist1 a b c => dist1B
      | dist2 a b c => dist2B
    end.

 (* the following requires specializing to the topos of sets, and must be assumed *)
 Conjecture must_assume: forall
  (t1 : ty)
  (t2 : ty)
  (e : exp (prod t1 t2) prop)
  (H0 : di (denote (curry e)))
  (H1 : hdi e)
  (D : Set)
  (I : inst t1 D),
  Included _ (fun J => denote e D (I, J)) (fun J => denoteB UB D I J).

Theorem semPres_holToNrc {G t} : forall (e: exp G t),
   hdi e -> denote e = denoteB (holToNrc e).
```

3.14 Appendix

3.14.1 Basic Lemmas

Lemma (PUSH-APPLY)**.**

$$apply; \langle f, g\rangle = \langle apply; f, apply; g\rangle \qquad \langle f, g\rangle; apply = \langle f; apply, g; apply\rangle$$

Proof.

- First we show

  $$apply; \langle f, g\rangle = \langle apply; f, apply; g\rangle$$

  We know that

  $$\langle f, g\rangle; \pi_1 = f \qquad \langle f, g\rangle; \pi_2 = g$$

  and indeed,

  $$apply; \langle f, g\rangle; \pi_1 = apply; f \qquad apply; \langle f, g\rangle; \pi_2 = apply; g$$

  rewriting our goal gives a new goal

  $$apply; \langle f, g\rangle = \langle apply; \langle f, g\rangle; \pi_1, apply; \langle f, g\rangle; \pi_2\rangle$$

  which follows from strong pairing (i.e., $x = \langle x; \pi_1, x; \pi_2\rangle$).

- Next we show

  $$\langle f, g\rangle; apply = \langle f; apply, g; apply\rangle$$

  By definition of apply, we may instead show

  $$\langle f, g\rangle; \langle \pi_1; apply, \pi_2; apply\rangle = \langle f; apply, g; apply\rangle$$

  From the bullet point above, we may instead show the following, which is easy:

  $$\langle \langle f, g\rangle; \pi_1; apply, \langle f, g\rangle; \pi_2; apply\rangle = \langle f; apply, g; apply\rangle$$

□

Lemma (PROJ-DI)**.** *Projection is domain-independent.*

Proof. We must show that $apply_{A\times B}; \pi_1 = \pi_1; apply_A$. By definition of $apply$, we know that $apply_{A\times B} = \langle \pi_1; apply_A, \pi_2; apply_B \rangle$. Substituting into our goal gives a new goal of $\langle \pi_1; apply_A, \pi_2; apply_B \rangle; \pi_1 = \pi_1; apply_A$, which follows by definition of π_1. □

Lemma (IN-MONO)**.** *In the topos of sets,*

$$f; inj_1 = g; inj_1 \quad implies \quad f = g \qquad\qquad f; inj_2 = g; inj_2 \quad implies \quad f = g$$

Proof. The maps $x \mapsto^{inj_1} (x, 0)$ and $x \mapsto^{inj_2} (x, 1)$ are bijections, and hence injective; in Set, injective functions are monomorphisms. □

Lemma (PAIR-INJ)**.**

$$\langle f, g \rangle = \langle f', g' \rangle \quad implies \quad f = f' \text{ and } g = g'$$

Proof. From $\langle f, g \rangle = \langle f', g' \rangle$ we have $\langle f, g \rangle; \pi_1 = \langle f', g' \rangle; \pi_1$ and hence $f = f'$. □

Lemma ($\downarrow$-INJ)**.**

$$\downarrow f = \downarrow g \quad implies \quad f = g$$

Proof. $\downarrow$ is a bijection $Hom(\llbracket s \rrbracket, \llbracket t \rrbracket) \cong Hom(\llbracket [s] \rrbracket, \llbracket [t] \rrbracket)$ □

Lemma (Λ-INJ)**.**

$$\Lambda f = \Lambda g \quad implies \quad f = g$$

Proof. In a CCC, Λ is a bijection $Hom(A \times B, C) \cong Hom(A, C^B)$. □

Lemma (ABS-IND-HELPER)**.** *Let $f : G \times T \to 2$ and $h : T \to S$ be morphisms in the topos of sets. Writing $\mathcal{P}(X) := 2^X$ for the exponential functor and $f \times g := \langle \pi_1; f, \pi_2; g \rangle$,*

$$(id \times h); (\Lambda f; P(h)) \times id; app = f$$

Proof. Suppose $(g, t) \mapsto f(g, t)$ is in $f : G \times T \to 2$. Then $g \mapsto \{(t, f(g, t)) | t \in T\}$ is in $\Lambda f : G \to 2^T$. Then $g \mapsto \{(h(t), f(g, t)) | t \in T\}$ is in $\Lambda f; P(h) : G \to 2^S$. Then $(g, t) \mapsto (\{(h(t), f(g, t)) | t \in T\}, h(t))$ is in $\Lambda f; P(h) \times h : G \times T \to 2^S \times S$. Then $(g, t) \mapsto f(g, t)$ is in $\Lambda f; P(h) \times h; ev : G \times T \to 2$. □

Lemma (APP-HELPER)**.** *In the topos of sets,*

$$\langle [\![[\Gamma] \vdash f' : [t] \to [2]]\!], [\![[\Gamma] \vdash e' : [t]]\!] \rangle ; \downarrow ev \;=\; [\![[\Gamma] \vdash e' \; mem \; f' : [2]]\!]$$

Proof. Follows by PRESERVE and noting that, in the topos of sets, both the evaluation map ev and the meaning of the NRC expression mem denote actual set membership; e.g., the binary predicate $\in$. □

3.14.2 Applicability of Inductive Hypothesis

Domain independence is logical for introduction rules, but *not* logical for elimination rules:

- Case PROJ1. Counterexample: $- \vdash (\lambda x{:}t.\bot, \lambda x{:}t.\top).1$ is domain-independent, but $- \vdash (\lambda x{:}t.\bot, \lambda x{:}t.\top)$ is not.
- Case CASE. Similar to above.
- Case EQ. Counterexample: $- \vdash \lambda x{:}t.\top = \lambda x{:}t.\top$.
- Case APP. Counterexample: $- \vdash (\lambda x{:}t \to 2.\top)(\lambda y{:}t.\top) : 2$ .This is equivalent to the constant morphism $true : 1 \to 2$, but neither sub-expression is domain-independent.

Lemma (VAR2-IND)**.**

$$di([\![\Gamma, y : s \vdash x{:}t]\!]) \quad implies \quad di([\![\Gamma \vdash x{:}t]\!])$$

Proof. We need not use antecedent, as $[\![\Gamma \vdash x{:}t]\!]$ is a composition of projections, and projection is domain-independent (PROJ-DI), and composition preserves domain independence (DI-COMP). □

Lemma (PAIR-IND)**.**

$$di([\![\Gamma \vdash (e, f) : s \times t]\!]) \qquad implies \qquad di([\![\Gamma \vdash e : s]\!]) \quad and \quad di([\![\Gamma \vdash f : t]\!])$$

Proof. Because $[\![\Gamma \vdash (e, f) : s \times t]\!]$ is domain-independent, we know (DI-SEM) that for every $\varphi : \mathbf{D_1} \hookrightarrow \mathbf{D_2}$,

$$apply^{\varphi}_{[\![\Gamma]\!]}; [\![\Gamma \vdash (e, f) : s \times t]\!]_{\mathbf{D_2}} = [\![\Gamma \vdash (e, f) : s \times t]\!]_{\mathbf{D_1}}; apply^{\varphi}_{[\![t]\!]}$$

Re-writing by PAIR-SEM gives

$$apply^{\varphi}_{[\![\Gamma]\!]}; \langle [\![\Gamma \vdash e : s]\!]_{\mathbf{D_2}}, [\![\Gamma \vdash f : t]\!]_{\mathbf{D_2}} \rangle = \langle [\![\Gamma \vdash e : s]\!]_{\mathbf{D_1}}, [\![\Gamma \vdash f : t]\!]_{\mathbf{D_1}} \rangle; apply^{\varphi}_{[\![t]\!]}$$

Re-writing by PUSH-APPLY gives

$$\langle apply^{\varphi}_{[\![\Gamma]\!]}; [\![\Gamma \vdash e : s]\!]_{\mathbf{D_2}}, apply^{\varphi}_{[\![\Gamma]\!]}; [\![\Gamma \vdash f : t]\!]_{\mathbf{D_2}} \rangle = \langle [\![\Gamma \vdash e : s]\!]_{\mathbf{D_1}}; apply^{\varphi}_{[\![t]\!]}, [\![\Gamma \vdash f : t]\!]_{\mathbf{D_1}}; apply^{\varphi}_{[\![t]\!]} \rangle$$

The result then follows from PAIR-INJ. □

Lemma (INJ-IND)**.**

$$di([\![\Gamma \vdash inl_t\ e : s + t]\!]) \quad implies \quad di([\![\Gamma \vdash e : s]\!])$$

Proof. Because $[\![\Gamma \vdash inl_t\ e : s + t]\!]$ is domain-independent, we know (DI-SEM) that for every $\varphi : \mathbf{D_1} \hookrightarrow \mathbf{D_2}$,

$$apply^{\varphi}_{\Gamma}; [\![\Gamma \vdash inl_t\ e : s + t]\!]_{\mathbf{D_2}} = [\![\Gamma \vdash inl_t\ e : s + t]\!]_{\mathbf{D_1}}; apply^{\varphi}_{s+t}$$

Re-writing by INL-SEM, we have

$$apply^{\varphi}_{\Gamma}; [\![\Gamma \vdash e : s]\!]_{\mathbf{D_2}}; inj_1 = [\![\Gamma \vdash e : s]\!]_{\mathbf{D_1}}; inj_1; apply^{\varphi}_{s+t}$$

By definition of apply,

$$apply^{\varphi}_{\Gamma}; [\![\Gamma \vdash e : s]\!]_{\mathbf{D_2}}; inj_1 = [\![\Gamma \vdash e : s]\!]_{\mathbf{D_1}}; inj_1; \langle apply^{\varphi}_{s}; inj_1 \oplus apply^{\varphi}_{t}; inj_2 \rangle$$

By SUM-BETA,

$$apply^{\varphi}_{\Gamma}; [\![\Gamma \vdash e : s]\!]_{\mathbf{D_2}}; inj_1 = [\![\Gamma \vdash e : s]\!]_{\mathbf{D_1}}; apply^{\varphi}_{s}; inj_1$$

The result then follows from IN-MONO. □

Lemma (ABS-IND)**.** *In the topos of sets,*

$$di(\downarrow [\![\Gamma \vdash \lambda x{:}t.e : t \to 2]\!]) \quad implies \quad di(\downarrow [\![\Gamma, x{:}t \vdash e : 2]\!])$$

Proof. We will prove a slightly more general result. Let $f_D : [\![G \times t]\!]_D \to 2$ be a family of morphisms indexed by D. If Λf_D is domain-independent, then so is f_D. For expediency we extend the *apply* operation to work over HOL types $t \to 2$, which is trivial because $[\![t \to 2]\!] \cong [\![Pt]\!]$.

We start with the statement of domain independence for Λf:

$$apply_G; \Lambda f_{D_2} = \Lambda f_{D_1}; apply_{2^t}$$

Applying the product functor and composing with *app* yields (recall that $f \times g := \langle \pi_1; f, \pi_2; g \rangle$):

$$apply_G \times id; \Lambda f_{D_2} \times id; app = \Lambda f_{D_1} \times id; apply_{2^t} \times id; app$$

β reduction then yields

$$apply_G \times id; f_{D_2} = \Lambda f_{D_1} \times id; apply_{2^t} \times id; app$$

composing with $id \times apply_t$ yields

$$id \times apply_t; apply_G \times id; f_{D_2} = id \times apply_t; \Lambda f_{D_1} \times id; apply_{2^t} \times id; app$$

and rearranging yields

$$apply_G \times apply_t; f_{D_2} = (\Lambda f_{D_1}; apply_{2^t}) \times apply_t; app$$

by definition of *apply*, this yields

$$apply_{G \times t}; f_{D_2} = (\Lambda f_{D_1}; apply_{2^t}) \times apply_t; app \qquad (known)$$

Because our goal is to prove that

$$apply_{G \times t}; f_{D_2} = f_{D_1}; apply_2$$

and $apply_2$ is the identity, it suffices by (known) to show

$$(\Lambda f_{D_1}; apply_{2^t}) \times apply_t; app = f_{D_1} \qquad (goal)$$

where (goal) is proved as a separate lemma (ABS-IND-HELPER).

□

3.14.3 Semantics Preservation, Other Cases

We now proves the semantic equivalence of the identity part of our translations HOL to NRC and NRC to HOL (by replacing $\downarrow$ with $\uparrow$ and $[]$ by $[]^{-1}$). We also prove the APP case here. ABS is inline in the text. Note that this only works for hereditarily domain-independent HOL expressions (it is easy to show that all NRC expressions are domain-independent).

Lemma. *Suppose* $\Gamma \vdash e : t$ *is a HOL typing derivation. Let* $\mathcal{T}$ *be a topos. Then for every object* $\mathbf{D}$ *in* $\mathcal{T}$

$$\downarrow_{\mathbf{D}} [\![\Gamma \vdash e : t]\!]_{\mathbf{D}} = [\![[\Gamma \vdash e : t]]\!]_{\mathbf{D}}$$

Proof. By induction on $\Gamma \vdash e : t$. First, we tackle the introduction rules:

- Case VAR1. Suppose $\Gamma \vdash e : t$ is of the form

$$\text{VAR1}\ \frac{}{\Gamma, x{:}t \vdash x{:}t} \qquad \text{VAR1-TRANS}\ \frac{}{[\Gamma, x{:}t \vdash x{:}t] = [\Gamma], x{:}[t] \vdash x{:}[t]} \qquad \text{VAR1-SEM}\ \frac{}{[\![\Gamma, x{:}t \vdash x{:}t]\!] = \pi_1^{[\![\Gamma]\!],[\![t]\!]}}$$

  We wish to show that

$$\downarrow [\![\Gamma, x{:}t \vdash x{:}t]\!] = [\![[\Gamma, x{:}t \vdash x{:}t]]\!]$$

  We calculate:

$$\begin{aligned}
&\downarrow [\![\Gamma, x{:}t \vdash x{:}t]\!] \\
&\qquad = \quad VAR1 - SEM \\
&\downarrow \pi_2^{[\![\Gamma]\!],[\![t]\!]} \\
&\qquad = \quad PRESERVE \\
&\pi_2^{[\![[\Gamma]]\!],[\![[t]]\!]} \\
&\qquad = \quad VAR1 - SEM \\
&\downarrow [\![[\Gamma], x{:}[t] \vdash x{:}[t]]\!] \\
&\qquad = \quad VAR1 - TRANS \\
&[\![[\Gamma, x{:}t \vdash x{:}t]]\!]
\end{aligned}$$

- Case VAR2. Suppose $\Gamma \vdash e : t$ is of the form

$$\frac{\Gamma \vdash x{:}t}{\Gamma, y : s \vdash x{:}t}\ \text{VAR2} \qquad \frac{[\Gamma \vdash x{:}t] = [\Gamma] \vdash x' : [t]}{[\Gamma, y : s \vdash x{:}t] = [\Gamma], y : [s] \vdash x' : [t]}\ \text{VAR2-TRANS} \qquad \frac{}{[\![\Gamma, y : s \vdash x{:}t]\!] = \pi_1^{[\![\Gamma]\!],[\![s]\!]}; [\![\Gamma \vdash x{:}t]\!]}\ \text{VAR2-SEM}$$

Our inductive hypothesis is

$$di([\![\Gamma \vdash x{:}t]\!]) \text{ implies } \downarrow [\![\Gamma \vdash x{:}t]\!] = [\![[\Gamma \vdash x{:}t]]\!]$$

We wish to show that

$$\downarrow [\![\Gamma, y : s \vdash x{:}t]\!] = [\![[\Gamma, y : s \vdash x{:}t]]\!]$$

We calculate:

$$\begin{aligned}
&\downarrow [\![\Gamma, y : s \vdash x{:}t]\!] \\
&\qquad = \quad VAR2 - SEM \\
&\downarrow (\pi_1; [\![\Gamma \vdash x{:}t]\!]) \\
&\qquad = \quad FUNCTOR \\
&(\downarrow \pi_1); (\downarrow [\![\Gamma \vdash x{:}t]\!]) \\
&\qquad = \quad PRESERVE \\
&\pi_1; (\downarrow [\![\Gamma \vdash x{:}t]\!]) \\
&\qquad = \quad IH \text{ with } VAR2 - IND \\
&\pi_1; [\![[\Gamma \vdash x{:}t]]\!] \\
&\qquad = \quad VAR2 - TRANS \\
&\pi_1; [\![[\Gamma] \vdash x' : [t]]\!] \\
&\qquad = \quad VAR2 - SEM \\
&[\![[\Gamma], y : [s] \vdash x' : [t]]\!] \\
&\qquad = \quad VAR2 - TRANS \\
&[\![[\Gamma, y : s \vdash x{:}t]]\!]
\end{aligned}$$

- Case UNIT. Suppose $\Gamma \vdash e : t$ is of the form

$$\text{UNIT}\quad \frac{}{\Gamma \vdash () : 1} \qquad \text{UNIT-TRANS}\quad \frac{}{[\Gamma \vdash () : 1] = [\Gamma] \vdash () : 1} \qquad \text{UNIT-SEM}\quad \frac{}{[\![\Gamma \vdash () : 1]\!] = \star_{[\![\Gamma]\!]}}$$

We wish to show that

$$\downarrow [\![\Gamma \vdash () : 1]\!] = [\![[\Gamma \vdash () : 1]]\!]$$

We calculate:

$$\begin{array}{rcl} \downarrow [\![\Gamma \vdash () : 1]\!] & & \\ & = & UNIT - SEM \\ \downarrow \star_{[\![\Gamma]\!]} & & \\ & = & PRESERVE \\ \star_{[\![[\Gamma]]\!]} & & \\ & = & UNIT - SEM \\ [\![[\Gamma] \vdash () : 1]\!] & & \\ & = & UNIT - TRANS \\ [\![[\Gamma \vdash () : 1]]\!] & & \end{array}$$

- Case PAIR. Suppose $\Gamma \vdash e : t$ is of the form

$$\text{PAIR}\quad \frac{\Gamma \vdash e : s \qquad \Gamma \vdash f : t}{\Gamma \vdash (e, f) : s \times t} \qquad \text{PAIR-TRANS}\quad \frac{[\Gamma \vdash e : s] = [\Gamma] \vdash e' : [s] \qquad [\Gamma \vdash f : t] = [\Gamma] \vdash f' : [t]}{[\Gamma \vdash (e, f) : s \times t] := [\Gamma] \vdash (e', f') : [s] \times [t]}$$

$$\text{PAIR-SEM}\quad \frac{\Gamma \vdash e : s \qquad \Gamma \vdash f : t}{[\![\Gamma \vdash (e, f) : s \times t]\!] := \langle [\![\Gamma \vdash e : s]\!], [\![\Gamma \vdash f : t]\!] \rangle}$$

Our inductive hypotheses are

$$di([\![\Gamma \vdash e : s]\!]) \text{ implies } \downarrow [\![\Gamma \vdash e : s]\!] = [\![[\Gamma \vdash e : s]]\!]$$

and

$$di(\llbracket \Gamma \vdash f : t \rrbracket) \text{ implies } \downarrow \llbracket \Gamma \vdash f : t \rrbracket = \llbracket [\Gamma \vdash f : t] \rrbracket$$

We wish to show that

$$\downarrow \llbracket \Gamma \vdash (e, f) : s \times t \rrbracket = \llbracket [\Gamma \vdash (e, f) : s \times t] \rrbracket$$

We calculate:

$$\begin{aligned}
&\downarrow \llbracket \Gamma \vdash (e, f) : s \times t \rrbracket \\
&\qquad = \quad PAIR - SEM \\
&\downarrow \langle \llbracket \Gamma \vdash e : s \rrbracket, \llbracket \Gamma \vdash f : t \rrbracket \rangle \\
&\qquad = \quad PUSH - \downarrow \\
&\langle \downarrow \llbracket \Gamma \vdash e : s \rrbracket, \downarrow \llbracket \Gamma \vdash f : t \rrbracket \rangle \\
&\qquad = \quad IH \\
&\langle \llbracket [\Gamma \vdash e : s] \rrbracket, \llbracket [\Gamma \vdash f : t] \rrbracket \rangle \\
&\qquad = \quad PAIR - TRANS \\
&\langle \llbracket [\Gamma] \vdash e' : [s] \rrbracket, \llbracket [\Gamma] \vdash f' : [t] \rrbracket \rangle \\
&\qquad = \quad PAIR - SEM \\
&\llbracket [\Gamma] \vdash (e', f') : [s] \times [t] \rrbracket \\
&\qquad = \quad PAIR - TRANS \\
&\llbracket [\Gamma \vdash (e, f) : s \times t] \rrbracket
\end{aligned}$$

- Case INL. Suppose $\Gamma \vdash e : t$ is of the form

$$\text{INL}\ \frac{\Gamma \vdash e : s}{\Gamma \vdash inl_t\ e : s + t} \qquad \text{INL-SEM}\ \frac{\Gamma \vdash e : s}{\llbracket \Gamma \vdash inl_t\ e : s + t \rrbracket := \llbracket \Gamma \vdash e : s \rrbracket; inj_1}$$

$$\text{INL-TRANS}\ \frac{[\Gamma \vdash e : s] = [\Gamma] \vdash e' : [s]}{[\Gamma \vdash inl_t\ e : s + t] := [\Gamma] \vdash inl_{[t]}\ e' : [s] + [t]}$$

Our inductive hypothesis is

$$di(\llbracket \Gamma \vdash e : s \rrbracket) \; implies \; \downarrow \llbracket \Gamma \vdash e : s \rrbracket = \llbracket [\Gamma \vdash e : s] \rrbracket$$

We wish to show that

$$\downarrow \llbracket \Gamma \vdash inl_t \; e : s + t \rrbracket = \llbracket [\Gamma \vdash inl_t \; e : s + t] \rrbracket$$

We calculate:

$$\begin{aligned}
&\downarrow \llbracket \Gamma \vdash inl_t \; e : s + t \rrbracket \\
&\qquad = \quad INL - SEM \\
&\downarrow (\llbracket \Gamma \vdash e : s \rrbracket; inj_1) \\
&\qquad = \quad FUNCTOR \\
&(\downarrow \llbracket \Gamma \vdash e : s \rrbracket); (\downarrow inj_1) \\
&\qquad = \quad PRESERVE \\
&(\downarrow \llbracket \Gamma \vdash e : s] \rrbracket); inj_1 \\
&\qquad = \quad IH \text{ with } INL - IND \\
&\llbracket [\Gamma \vdash e : s] \rrbracket; inj_1 \\
&\qquad = \quad INL - TRANS \\
&\llbracket [\Gamma] \vdash e' : [s] \rrbracket; inj_1 \\
&\qquad = \quad INL - SEM \\
&\llbracket [\Gamma] \vdash inl_{[t]} \; e' : [s] + [t] \rrbracket \\
&\qquad = \quad INL - TRANS \\
&\llbracket [\Gamma \vdash inl_t \; e : s + t] \rrbracket
\end{aligned}$$

- Case INR. Similar to INL

Next, we tackle the elimination rules, which can only invoke their IHs because of the hereditarily domain-independence requirement.

- Case VOID. Suppose $\Gamma \vdash e : t$ is of the form

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \mathit{ff}\ e : t}\ \text{VOID} \qquad \frac{[\Gamma \vdash e : 0] = [\Gamma] \vdash e' : [t]}{[\Gamma \vdash \mathit{ff}\ e : 1] := [\Gamma] \vdash \mathit{ff}\ e' : [t]}\ \text{VOID-TRANS} \qquad \frac{}{[\![\Gamma \vdash \mathit{ff}\ e : t]\!] = [\![\Gamma \vdash e : 0]\!]; \mathit{ff}^{[\![t]\!]}}\ \text{VOID-SEM}$$

We wish to show that

$$\downarrow [\![\Gamma \vdash \mathit{ff}\ e : t]\!] = [\![[\Gamma \vdash \mathit{ff}\ e : t]]\!]$$

Our inductive hypothesis is that

$$\downarrow [\![\Gamma \vdash e]\!] : 0 = [\![[\Gamma \vdash e : 0]]\!]$$

We calculate:

$$\begin{aligned}
&\downarrow [\![\Gamma \vdash \mathit{ff}\ e : t]\!] \\
&\qquad = \quad VOID - SEM \\
&\downarrow [\![\Gamma \vdash e : 0]\!]; \mathit{ff}^{[\![t]\!]} \\
&\qquad = \quad FUNCTOR \\
&(\downarrow [\![\Gamma \vdash e : 0]\!]); (\downarrow \mathit{ff}^{[\![t]\!]}) \\
&\qquad = \quad IH \\
&[\![[\Gamma \vdash e : 0]]\!]; \downarrow \mathit{ff}^{[\![t]\!]} \\
&\qquad = \quad PRESERVE \\
&[\![[\Gamma \vdash e : 0]]\!]; \mathit{ff}^{[\![[t]]\!]} \\
&\qquad = \quad VOID - TRANS \\
&[\![[\Gamma] \vdash e' : [0]]\!]; \mathit{ff}^{[\![[0]]\!]} \\
&\qquad = \quad VOID - SEM \\
&[\![[\Gamma] \vdash \mathit{ff}\ e' : [t]]\!] \\
&\qquad = \quad VOID - TRANS \\
&[\![[\Gamma \vdash \mathit{ff}\ e : t]]\!]
\end{aligned}$$

- Case PROJ1. Suppose $\Gamma \vdash e : t$ is of the form

$$\text{PROJ1}\ \frac{\Gamma \vdash e : s \times t}{\Gamma \vdash e.1 : s} \qquad \text{PROJ1-TRANS}\ \frac{[\Gamma \vdash e : s \times t] = [\Gamma] \vdash e' : [s] \times [t]}{[\Gamma \vdash e.1 : s] = [\Gamma] \vdash e'.1 : [s]} \qquad \text{PROJ1-SEM}\ \frac{\Gamma \vdash e : s \times t}{[\![\Gamma \vdash e.1 : s]\!] := [\![\Gamma \vdash e : s \times t]\!]; \pi_1}$$

Our inductive hypothesis is that

$$di([\![\Gamma \vdash e : s \times t]\!]) \text{ implies } \downarrow [\![\Gamma \vdash e : s \times t]\!] = [\![[\Gamma \vdash e : s \times t]]\!]$$

We wish to show that

$$\downarrow [\![\Gamma \vdash e.1 : s]\!] = [\![[\Gamma \vdash e.1 : s]]\!]$$

We calculate:

$$\begin{aligned}
&\downarrow [\![\Gamma \vdash e.1 : s]\!] \\
&\qquad = \quad PROJ1 - SEM \\
&\downarrow ([\![\Gamma \vdash e : s \times t]\!]; \pi_1) \\
&\qquad = \quad FUNCTOR \\
&(\downarrow [\![\Gamma \vdash e : s \times t]\!]); (\downarrow \pi_1) \\
&\qquad = \quad PRESERVE \\
&\downarrow [\![\Gamma \vdash e : s \times t]\!]; \pi_1 \\
&\qquad = \quad IH \text{ with } PROJ1 - IND \\
&[\![[\Gamma \vdash e : s \times t]]\!]; \pi_1 \\
&\qquad = \quad PROJ1 - TRANS \\
&[\![[\Gamma] \vdash e' : [s] \times [t]]\!]; \pi_1 \\
&\qquad = \quad PROJ1 - SEM \\
&[\![[\Gamma] \vdash e'.1 : [s]]\!] \\
&\qquad = \quad PROJ1 - TRANS \\
&[\![[\Gamma \vdash e.1 : s]]\!]
\end{aligned}$$

- Case PROJ2. Similar to PROJ1.

- Case CASE. Suppose $\Gamma \vdash e : t$ is of the form

$$\text{CASE}\quad \frac{\Gamma \vdash e : s + t \qquad \Gamma, x{:}s \vdash f : u \qquad \Gamma, y{:}t \vdash g : u}{\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g\ : u}$$

$$\text{CASE-SEM}\quad \frac{\Gamma \vdash e : s + t \qquad \Gamma, x{:}s \vdash f : u \qquad \Gamma, y{:}t \vdash g : u}{\begin{array}{c}[\![\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{else}\ \lambda y{:}t.g : u]\!] := \\ \langle \mathit{id}, [\![\Gamma \vdash e : s + t]\!] \rangle; \mathit{dist}; \langle [\![\Gamma, x{:}s \vdash f : u]\!] \oplus [\![\Gamma, y{:}t \vdash g : u]\!] \rangle\end{array}}$$

$$\text{CASE-TRANS}\quad \frac{[\Gamma \vdash e : s + t] = [\Gamma] \vdash e' : [s] + [t] \qquad [\Gamma, x{:}s \vdash f : u] = [\Gamma], x{:}[s] \vdash f' : [u] \qquad [\Gamma, y{:}t \vdash g : u] = [\Gamma], y{:}[t] \vdash g' : [u]}{[\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g\ : u] := [\Gamma] \vdash \mathit{case}\ e'\ \mathit{of}\ \lambda x{:}[s].f'\ \mathit{or}\ \lambda y{:}[t].g' : [u]}$$

Our inductive hypothesis is

$$di(\downarrow [\![\Gamma \vdash e : s + t]\!])\ \mathit{implies}\ \downarrow [\![\Gamma \vdash e : s + t]\!] = [\![[\Gamma \vdash e : s + t]]\!]$$

and

$$di(\downarrow [\![\Gamma, x{:}s \vdash f : u]\!])\ \mathit{implies}\ \downarrow [\![\Gamma, x{:}s \vdash f : u]\!] = [\![[\Gamma, x{:}s \vdash f : u]]\!]$$

and

$$di(\downarrow [\![\Gamma, y{:}t \vdash g : u]\!])\ \mathit{implies}\ \downarrow [\![\Gamma, y{:}t \vdash g : u]\!] = [\![[\Gamma, y{:}t \vdash g : u]]\!]$$

We wish to show that

$$\downarrow [\![\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g\ : u]\!] = [\![[\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g\ : u]]\!]$$

We calculate:

$$
\begin{aligned}
& \downarrow [\![\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g\ : u]\!] \\
CASE-SEM \quad &= \\
& \downarrow \big(\langle \mathit{id}, [\![\Gamma \vdash e : s+t]\!]\rangle; \mathit{dist}; \langle [\![\Gamma, x{:}s \vdash f : u]\!] \oplus [\![\Gamma, y{:}t \vdash g : u]\!]\rangle\big) \\
FUNCTOR \quad &= \\
& \downarrow \langle \mathit{id}, [\![\Gamma \vdash e : s+t]\!]\rangle; (\downarrow \mathit{dist}); \downarrow \langle [\![\Gamma, x{:}s \vdash f : u]\!] \oplus [\![\Gamma, y{:}t \vdash g : u]\!]\rangle \\
PUSH{-}\downarrow \quad &= \\
& \langle \downarrow \mathit{id}, \downarrow [\![\Gamma \vdash e : s+t]\!]\rangle; (\downarrow \mathit{dist}); \langle \downarrow [\![\Gamma, x{:}s \vdash f : u]\!] \oplus \downarrow [\![\Gamma, y{:}t \vdash g : u]\!]\rangle \\
PRESERVE \quad &= \\
& \langle \mathit{id}, \downarrow [\![\Gamma \vdash e : s+t]\!]\rangle; \mathit{dist}; \langle \downarrow [\![\Gamma, x{:}s \vdash f : u]\!] \oplus \downarrow [\![\Gamma, y{:}t \vdash g : u]\!]\rangle\big) \\
IH\ and\ CASE-IND \quad &= \\
& \langle \mathit{id}, [\![[\Gamma \vdash e : s+t]]\!]\rangle; \mathit{dist}; \langle [\![[\Gamma, x{:}s \vdash f : u]]\!] \oplus [\![[\Gamma, y{:}t \vdash g : u]]\!]\rangle\big) \\
CASE-TRANS \quad &= \\
& \langle \mathit{id}, [\![[\Gamma] \vdash e' : [s]+[t]]\!]\rangle; \mathit{dist}; \langle [\![[\Gamma], x{:}[s] \vdash f : [u]]\!] \oplus [\![[\Gamma], y{:}[t] \vdash g : [u]]\!]\rangle \\
CASE-SEM \quad &= \\
& [\![[\Gamma] \vdash \mathit{case}\ e'\ \mathit{of}\ \lambda x{:}[s].f'\ \mathit{or}\ \lambda y{:}[t].g' : [u]]\!] \\
CASE-TRANS \quad &= \\
& [\![[\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ \lambda x{:}s.f\ \mathit{or}\ \lambda y{:}t.g\ : u]]\!]
\end{aligned}
$$

- Case EQ. Suppose $\Gamma \vdash e : t$ is of the form

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash f : t}{\Gamma \vdash e = f : t}$$

EQ-SEM

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash f : t}{[\![\Gamma \vdash e = f : 2]\!] := \delta \circ \langle [\![\Gamma \vdash e : t]\!], [\![\Gamma \vdash f : t]\!]\rangle}$$

EQ-TRANS

$$\frac{[\Gamma \vdash e : t] = [\Gamma] \vdash e' : [t] \qquad [\Gamma \vdash f : t] = [\Gamma] \vdash f' : [t]}{[\Gamma \vdash e = f : 2] := [\Gamma] \vdash e' = f' : 2}$$

Our inductive hypothesis is

$$di(\llbracket \Gamma \vdash e : t \rrbracket) \text{ implies } \downarrow \llbracket \Gamma \vdash e : t \rrbracket = \llbracket [\Gamma \vdash e : t] \rrbracket$$

and

$$di(\llbracket \Gamma \vdash f : t \rrbracket) \text{ implies } \downarrow \llbracket \Gamma \vdash f : t \rrbracket = \llbracket [\Gamma \vdash f : t] \rrbracket$$

we wish to show that

$$\downarrow \llbracket \Gamma \vdash e = f : 2 \rrbracket = \llbracket [\Gamma \vdash e = f : 2] \rrbracket$$

We calculate:

$$\begin{aligned}
&\downarrow \llbracket \Gamma \vdash e = f : 2 \rrbracket \\
&\qquad = \quad EQ - SEM \\
&\downarrow (\langle \llbracket \Gamma \vdash e : t \rrbracket, \llbracket \Gamma \vdash f : t \rrbracket \rangle; \delta) \\
&\qquad = \quad FUNCTOR \\
&(\downarrow \langle \llbracket \Gamma \vdash e : t \rrbracket, \llbracket \Gamma \vdash f : t \rrbracket \rangle); (\downarrow \delta) \\
&\qquad = \quad PRESERVE \\
&\langle \downarrow \llbracket \Gamma \vdash e : t \rrbracket, \downarrow \llbracket \Gamma \vdash f : t \rrbracket \rangle; \delta \\
&\qquad = \quad IH \text{ and } EQ - IND \\
&\langle \llbracket [\Gamma \vdash e : t] \rrbracket, \llbracket [\Gamma \vdash f : t] \rrbracket \rangle; \delta \\
&\qquad = \quad EQ - TRANS \\
&\langle \llbracket [\Gamma] \vdash e' : [t] \rrbracket, \llbracket [\Gamma] \vdash f' : [t] \rrbracket \rangle; \delta \\
&\qquad = \quad EQ - SEM \\
&\llbracket [\Gamma] \vdash e' = f' : 2 \rrbracket \\
&\qquad = \quad EQ - TRANS \\
&\llbracket [\Gamma \vdash e = f : 2] \rrbracket
\end{aligned}$$

- Case APP. Suppose $\Gamma \vdash e : t$ is of the form

$$\frac{\Gamma \vdash f : t \to 2 \qquad \Gamma \vdash e : t}{\Gamma \vdash f\ e : 2}$$

APP-SEM

$$[\![\Gamma \vdash fe : 2]\!] := \langle [\![\Gamma \vdash f : t \to 2]\!], [\![\Gamma \vdash e : t]\!] \rangle ; ev$$

APP-TRANS

$$\frac{[\Gamma \vdash f : t \to 2] = [\Gamma] \vdash f' : P[t] \qquad [\Gamma \vdash e : t] = [\Gamma] \vdash e' : [t]}{[\Gamma \vdash fe : 2] := [\Gamma] \vdash e'\ mem\ f' : [2]}$$

Our inductive hypotheses are that

$$di([\![\Gamma \vdash f : t \to 2]\!]) \text{ implies } \downarrow [\![\Gamma \vdash f : t \to 2]\!] = [\![[\Gamma \vdash f : t \to 2]]\!]$$

$$di([\![\Gamma \vdash e : t]\!]) \text{ implies } \downarrow [\![\Gamma \vdash e : t]\!] = [\![[\Gamma \vdash e : t]]\!]$$

We wish to show that

$$\downarrow [\![\Gamma \vdash fe : 2]\!] = [\![[\Gamma \vdash fe : 2]]\!]$$

We calculate:

$$\begin{aligned}
&\downarrow [\![\Gamma \vdash fe : 2]\!] \\
&\quad = \quad APP - SEM \\
&\downarrow \langle [\![\Gamma \vdash f : t \to 2]\!], [\![\Gamma \vdash e : t]\!] \rangle ; ev \\
&\quad = \quad PUSH - \downarrow \\
&\langle \downarrow [\![\Gamma \vdash f : t \to 2]\!], \downarrow [\![\Gamma \vdash e : t]\!] \rangle ; \downarrow ev \\
&\quad = \quad IH \\
&\langle [\![[\Gamma \vdash f : t \to 2]]\!], [\![[\Gamma \vdash e : t]]\!] \rangle ; \downarrow ev \\
&\quad = \quad APP - TRANS \\
&\langle [\![[\Gamma] \vdash f' : [t] \to [2]]\!], [\![[\Gamma] \vdash e' : [t]]\!] \rangle ; \downarrow ev \\
&\quad = \quad APP - HELPER \\
&[\![[\Gamma] \vdash e'\ mem\ f' : [2]]\!] \\
&\quad = \quad APP - TRANS \\
&[\![[\Gamma \vdash fe : 2]]\!]
\end{aligned}$$

Where APP-HELPER is proved as a separate lemma.

□

Chapter 4

Relational Foundations for Functorial Data Migration

The work in this chapter is joint with David Spivak, who developed the original mathematics of the functorial data model [74]. All proofs in this chapter were proved by David, and all software in this chapter was developed by me. The definition of FQL as a functional query language is my contribution, and all theorems, excluding the closure of Δ, Σ, Π under composition, were proved in service of compiling FQL to relational algebra and other relational languages.

4.1 Introduction

In this chapter we use the tools of category theory to develop a model theory and operator algebra for database instances that satisfy path equality constraints. In doing so we pick up a long line of work aptly summarized by Melnik in his thesis [61]. He describes three ways for achieving an "executable model theory" suitable for large-scale and generic information-integration efforts [62]; we quote these here:

1. One way is to consider schemas, instances, and mappings as syntactic objects represented in a common meta-theory, e.g., as graphs. This approach has been pursued in almost all prior work on generic model management. In essence, the operators are specified by means of graph transformations. As long as the graph transformations do not exploit any knowledge of what the

graphs actually represent, the operators can be considered truly generic. Unfortunately, there are very few useful operations that can be defined in such an agnostic fashion. Largely, they are limited to Subgraph, Copy, and the set operations on graphs.

2. A second way to achieve generic applicability is by using state-based semantics. In this approach, the properties of the operators are characterized in terms of instances of schemas that are taken as input and produced as output. Under the assumption that schemas possess well-defined sets of instances, all key operators can be characterized in a truly generic fashion. Such characterization is applicable to very complex kinds of schemas and mappings that are used in real applications, including XML Schemas, XQuery, and SQL. Although state-based characterization does not provide a detailed implementation blueprint, it is sufficiently specific so that the effect of the operators can be worked out for concrete languages.

3. A third way for addressing generic applicability is an axiomatic one, e.g., using a category-theoretic approach. The idea of the approach is to define the operators using axioms that are expressed in terms of the operators to be defined. Associativity of compose or commutativity of merge are examples of such axioms. This approach seems to be the most challenging, both in terms of determining a useful set of axioms and implementing the operators in such a way that the axioms hold when the operators are applied to concrete languages.

Whereas Melnik proceeds to develop approach 2, and to implement a prototype called Rondo [63], in this chapter we develop a new approach that combines 1 and 3. In particular, our schemas are *categories*, and our operators are those of the functorial data model [74]. Because they are categories, our schemas generalize graphs yet are significantly more expressive (approach 1); they also form a category, the category of categories, which is well understood categorically (approach 3). We believe that our combined approach enjoys the benefits of approaches 1 and 3 while nullifying their disadvantages.

4.1.1 Background

In the functorial data model [74], database schemas are finitely presented categories [9]: directed labeled multi-graphs with path-equivalence constraints. Database instances are functors from categories/schemas to the category of sets. By targeting the category of sets, database instances can be stored as relational tables. Database morphisms are natural transformations (morphisms of functors) from database instances

to database instances. The database instances and morphisms on a schema S constitute a category, denoted S–**Inst**. A morphism M between schemas S and T, which can be generated from a visual correspondence between graphs, induces three adjoint data migration functors, $\Sigma_M : S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$, $\Pi_M : S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$, and $\Delta_M : T\text{–}\mathbf{Inst} \to S\text{–}\mathbf{Inst}$. At a high-level, this functorial data model provides an alternative category-theoretic foundation from which to study problems in information management. The mathematical foundations of this data model are developed by Spivak in [74] using the language of category theory, but few specific connections are made to relational database theory.

4.1.2 Related Work

Although labeled mutli-graphs with path equivalence constraints are a common notation for schemas [16], there has been little work to treat such schemas categorically [35]. Instead, most schema transformation frameworks treat graphs as relational schemas. For example, in Clio [44], users draw lines connecting related elements between two schemas-as-graphs, and Clio generates a relational query that implements the user's intended data transformation. Behind the scenes, the user's correspondence is translated into a formula in the relational language of second-order tuple generating dependencies, from which a query is generated [32]. As another example, in the Rondo system [63], users are presented with an ad-hoc collection of operators over schema-as-graphs that they then script together to implement a data transformation. Although graphs and path-equivalences are used as inputs for both Clio and Rondo, both Clio and Rondo immediately translate from graphs and path-equivalences into a relational schema before proceeding further.

In many ways, our work is an extension and improvement of Rosebrugh et al's initial work on understanding category presentations as database schemas [35]. In that work, the authors identify the Σ and Π data migration functors, but they do not identify Δ as their adjoint. Moreover, they remark that they were unable to implement Σ and Π using relational algebra, and they do not formalize a query language or investigate the behavior of the Σ and Π with respect to composition. Our mathematical development diverges from their subsequent work on "sketches" [52].

4.1.3 Contributions and Outline

In this chapter we make the following contributions:

1. As described above, the functorial data model [74] is not quite appropriate for many practical information management tasks. Intuitively, this is because every instance in the pure functorial data model behaves like a relational database instance where all values are IDs, no values are constants, and equality is actually isomorphism. So, we extend the functorial data model with "attributes" to capture meaningful concrete data. The practical result is that our schemas become special kinds of entity-relationship (ER) diagrams [37], and our instances can be represented as relational tables that conform to such diagrams. (Sections 2 and 3)

2. We define a simple algebraic query language FQL where every query denotes a data migration functor in this new extended sense. We show that FQL is closed under composition and how every query in FQL can be written as three graph correspondences roughly corresponding to projection, join, and union. Determining whether three arbitrary graph correspondences are FQL queries is semi-decidable. (Section 4)

3. We provide a translation of FQL into SQL, by which we mean the union of two languages: 1) the SPCU relational algebra of selection, projection, cartesian product, and union, under its typical set semantics, and 2) a globally unique ID generator that constructs $N+1$-ary tables from N-ary tables by adding a globally unique ID to each row. This allows us to easily generate SQL programs that implement FQL. An immediate corollary is that materializing result instances of FQL queries has polynomial time data complexity. (Section 5)

4. We show that every union of relational conjunctive queries (SPCU) *under bag semantics* is expressible in FQL and how to extend FQL to capture every union of relational conjunctive queries under set semantics. (Section 6)

5. We show that FQL is a schema transformation framework in the sense of Alagic and Bernstein's categorical model theory [3] (Section 7).

6. We discuss the relationship between FQL and embedded dependencies (Section 8).

7. We implemented FQL in a prototype visual schema mapping and SQL generation tool in the spirit of Clio and Rondo, available at wisnesky.net/fql.html. We present a tutorial for the tool in section 9.

8. We conclude with a presentation of a FQL as a language of categorical combinators [24] (Section 10).

All the proofs referenced in this chapter may be found in a tech report [76]. For the remainder of the introduction we motivate the development of FQL by presenting a detailed comparison of FQL to Clio [44].

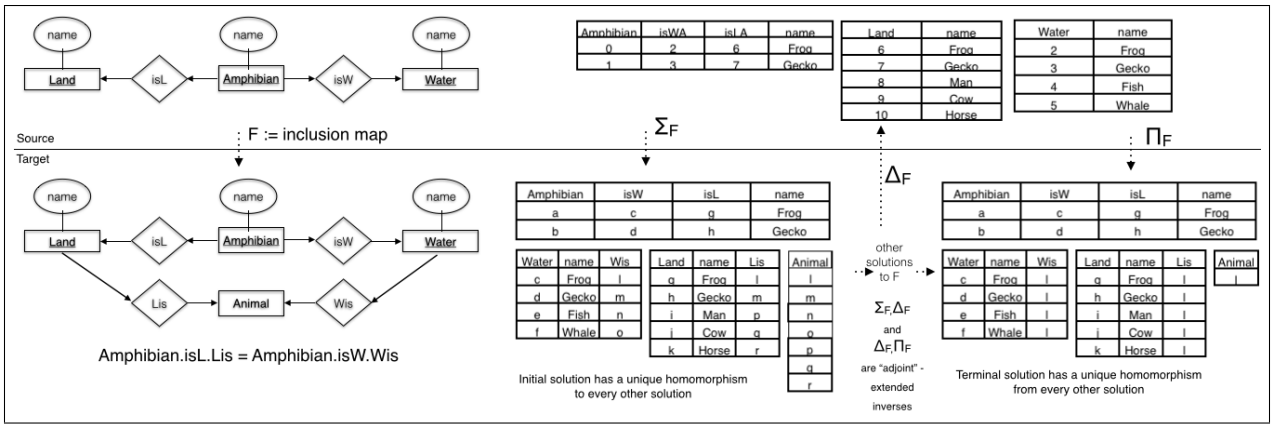


Figure 4.1: Example Data Migration Setting

4.1.4 Motivation

Functorial data migration has many desirable properties that traditional relational approaches such as employed by Clio [44] do not. We will now elaborate using an example. In the following, we treat Clio as a representative data migration tool that draws on three distinct areas of existing work: schema mapping generation from schema correspondences (e.g., [71]), schema mapping semantics (e.g., the data-exchange chase introduced in [30] that produces universal solutions), and schema mapping operators (e.g., extended inverses [33]). Other relational data migration tools (e.g., Rondo [63]) also draw on this body of work.

Consider the source entity-relationship (ER) diagram [37] in figure 4.1, with entities being amphibians, land animals, and water animals. All relationships are of the form "is a". The ER diagram defines a relational schema consisting of tables Amphibian, Land, and Water, each with a primary key column of the same name. Each table has a name column, and Amphibian has foreign key columns isL (to Land), and isW (to Water). The target ER diagram in figure 1 adds an additional entity, Animal, and adds foreign keys to Animal from tables Water (Wis) and Land (Lis). The diagram also contains an equation between paths of foreign keys, stating that each Amphibian is a single Animal, even though each Amphibian is both a Land animal and a Water animal. Let F denote the obvious inclusion mapping from the source schema to the target schema. Given such an F, schema mapping tools such as FQL and Clio will emit queries, usually in a variety of languages, for migrating source instances to the target schema. We will now contrast the behavior of Clio and FQL in this data migration setting.

Clio begins by translating the foreign key constraints into tuple-generating dependencies (TGDs). It uses the TGDs from each schema to compute maximal "connections" (also called "logical relations") among the tables within the schema, and then it uses the input mapping to compute a set of source-to-target TGDs

that relate source logical relations to target logical relations [71]. Given an input instance I, Clio solves the output TGDs by chasing with them on I [30]. The output instance $clio(I)$ will be a "universal solution" to (F, I), meaning that for any other solution J, there will be a homomorphism $clio(I) \to J$ [30].

FQL begins similarly. It treats the ER diagram as a graph, closes the graph under composition of paths, and then quotients this free graph by the path equalities. Then, from the mapping $F : S \to T$, FQL computes three data migration operations, $\Sigma_F \colon S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$, $\Pi_F \colon S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$, and $\Delta_F \colon T\text{–}\mathbf{Inst} \to S\text{–}\mathbf{Inst}$. Given an input instance I, $\Sigma_F(I)$ will be an *initial solution* to (I, F), meaning that for any other solution J, there will be a unique homomorphism $\Sigma_F(I) \to J$. Dually, $\Pi_F(I)$ will be a *terminal solution*, meaning that there will be a unique homomorphism $J \to \Pi_F(I)$. The Δ_F operation is the crucial reverse data migration operation identified by Bernstein and Alagic in their categorical model theory [3], and Σ_F and Π_F are Δ_F's unique left and right "adjoints", or weak inverses [9].

FQL has the following advantages over Clio in the data migration setting in figure 4.1:

- As path equality constraints such as in figure 4.1 translate into equality-generating dependencies (EGDs), not TGDs, Clio ignores them. Hence, Clio computes 9 animals in figure 1, instead of the 7 animals computed by FQL's Σ operator. Intuitively, Clio computes 9 animals because the path Amphibian.isL.Lis, when applied to "a", is instantiated by a different value than the path Amphibian.isW.Wis when applied to the same "a". FQL correctly identifies the two values, resulting in 7 animals, via the path equation.
- FQL's initial solution is stronger than Clio's universal solution because the mediating homomorphism is unique.
- FQL's Δ_F, Σ_F, and Π_F operations are unique. In Clio, many data migration queries can be generated from a schema mapping F.
- FQL's Σ_F is Δ_F's left adjoint, and Π_F is Δ_F's right adjoint. Adjoints are a weaker notion of inverse than the quasi-inverses of [33], but unlike quasi-inverses, FQL's Σ_F and Π_F always exist. When F is "fully-faithful" [9], Σ_F is the quasi-inverse of Δ_F.
- Clio lacks an operation corresponding to Π. Although Σ often captures the traditional semantics of data-migration, Π provides a significantly leap in expressive power: without Π, FQL would not be able to implement the SPCU relational algebra.

Of course, Clio has advantages over FQL. For example, FQL cannot currently handle nested data.

4.2 Categorical Data

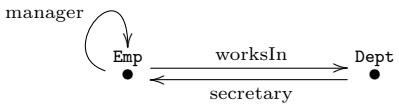
In this section we define the original signatures and instances of Spivak's [74], as well as "typed signatures" and "typed instances", which are our extension of Spivak's [74] to attributes. The basic idea is that signatures are stylized ER diagrams that denote categories, and our database instances can be represented as instances of such ER diagrams, and vice versa (up to natural isomorphism).

4.2.1 Signatures

The functorial data model [74] uses directed labeled multi-graphs and path equalities for signatures. A *path* p is defined inductively as:

$$p ::= node \mid p.edge$$

A *signature* S is a finite presentation of a category. That is, a signature S is a triple $S := (N, E, C)$ where N is a finite set of nodes, E is a finite set of labeled directed edges , and C a finite set of path equivalences. For example,



$$\texttt{Emp.manager.worksIn} = \texttt{Emp.worksIn} \qquad \texttt{Dept.secretary.worksIn} = \texttt{Dept}$$

Here we see a signature S with two vertices and three arrows, and two path equivalence statements. This information generates a category $[\![S]\!]$: the free category on the graph, modulo the equivalence relation induced by the path equivalences. The category $[\![S]\!]$ is the *schema* for S, and database instances over $[\![S]\!]$ are functors $[\![S]\!] \to \mathbf{Set}$. Every path $p\colon X \to Y$ in a signature S denotes a morphism $[\![p]\!]\colon [\![X]\!] \to [\![Y]\!]$ in $[\![S]\!]$. Given two paths p_1, p_2 in a signature S, we say that they are *equivalent*, written $p_1 \cong p_2$ if $[\![p_1]\!]$ and $[\![p_2]\!]$ are the same morphism in $[\![S]\!]$. Two signatures S and T are *isomorphic*, written $S \cong T$, if they denote isomorphic schema, i.e., if the categories they generate are isomorphic.

4.2.2 Cyclic Signatures

If a signature contains a loop, it may or may not denote a category with infinitely many morphisms. Hence, some constructions over signatures may not computable. Testing if two paths in a signature are equivalent is known as the *word problem* for categories. The word problem can be semi-decided using the "completion without failure" extension [6] of the Knuth-Bendix algorithm. This algorithm first attempts to construct a strongly normalizing re-write system based on the path equalities; if it succeeds, it yields a linear time decision procedure for the word problem [48]. If a signature denotes a finite category, the Carmody-Walters algorithm [18] will compute its denotation. The algorithm computes *left Kan extensions* and can be used for many other purposes in computational category theory [35]. In fact, every Σ functor arises as a left Kan extension, and vice versa.

4.2.3 Instances

Let S be a signature. A $[\![S]\!]$-instance is a functor from $[\![S]\!]$ to the category of sets. We will represent instances as relational tables using the following binary format:

- To each node N corresponds an "identity" or "entity" table named N, a reflexive table with tuples of the form (x, x). We can specify this using first-order logic:

$$\forall xy.N(x,y) \Rightarrow x = y. \tag{4.1}$$

  The entries in these tables are called *ID*s or *keys*, and for the purposes of this chapter we require them to be globally unique. We call this the *globally unique key assumption*.

- To each edge $e\colon N_1 \to N_2$ corresponds a "link" table e between identity tables N_1 and N_2. The axioms below merely say that every edge $e\colon N_1 \to N_2$ designates a total function $N_1 \to N_2$:

$$\begin{aligned}
&\forall xy.\ e(x,y) \Rightarrow N_1(x,x)\\
&\forall xy.\ e(x,y) \Rightarrow N_2(y,y)\\
&\forall xyz.\ e(x,y) \wedge e(x,z) \Rightarrow y = z\\
&\forall x.\ N_1(x,x) \Rightarrow \exists y.e(x,y)
\end{aligned} \tag{4.2}$$

An example instance of our employees schema is:

Emp	
Emp	**Emp**
101	101
102	102
103	103

Dept	
Dept	**Dept**
q10	q10
x02	x02

manager	
Emp	**Emp**
101	103
102	102
103	103

worksIn	
Emp	**Dept**
101	q10
102	q10
103	x02

secretary	
Dept	**Emp**
x02	102
q10	101

To save space, we will sometimes present instances in a "joined" format:

Emp		
Emp	**manager**	**worksIn**
101	103	q10
102	102	q10
103	103	x02

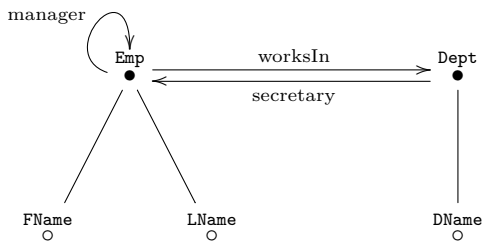
Dept	
Dept	**secretary**
q10	102
x02	101

The natural notion of equality of instances is *isomorphism*. In particular, the actual constants in the above tables should be considered meaningless IDs.

4.2.4 Attributes

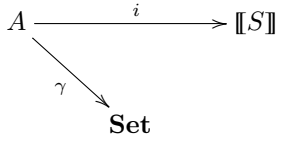
Signatures and instances, as defined above, do not have quite enough structure to be useful in practice. At a practical level, we usually need fixed atomic domains such as String and Nat to store actual data. Hence, in this section we extend the functorial data model with *attributes*.

Let S be a signature. A *typing* Γ for S is a mapping from each node N of S to a (possibly empty) set of attribute names and associated base types (Nat, String, etc), written $\Gamma(N)$. We call a pairing of a signature and a typing a *typed signature*. Borrowing from ER-diagram notation, we will write attributes as open circles. For example, we might enrich our previous signature with a typing as follows:



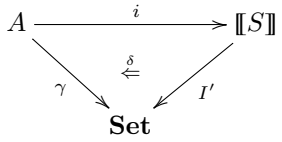
$\texttt{Emp.manager.worksIn} = \texttt{Emp.worksIn} \qquad \texttt{Dept.secretary.worksIn} = \texttt{Dept}$

Importantly, path expressions may not refer to attributes; they may only refer to nodes and directed edges. The meaning of Γ is a triple $[\![\Gamma]\!] := (A, i, \gamma)$, where A is a discrete category consisting of the attributes of Γ, i is a functor from A to $[\![S]\!]$ mapping each attribute to its corresponding node, and γ is a functor from A to Set, mapping each attribute to its domain (e.g., the set of strings, the set of natural numbers):



4.2.5 Typed Instances

Let S be a signature and Γ a typing such that $[\![\Gamma]\!] = (A, i, \gamma)$. A *typed instance* I is a pair (I', δ) where $I' \colon [\![S]\!] \to \mathbf{Set}$ is an (untyped) instance together with a natural transformation $\delta \colon I' \circ i \Rightarrow \gamma$. Intuitively, δ associates an appropriately typed constant (e.g., a string) to each globally unique ID in I':



We represent the δ-part of a typed instance as a set of binary tables as follows:

- To each node table N and attribute A with type t in $\Gamma(N)$ corresponds a binary "attribute" table mapping the domain of N to values of type t.

In our employees example, we might add the following:

FName	
Emp	**String**
101	Alan
102	Andrey
103	Camille

LName	
Emp	**String**
101	Turing
102	Markov
103	Jordan

DName	
Dept	**String**
x02	Math
q10	CS

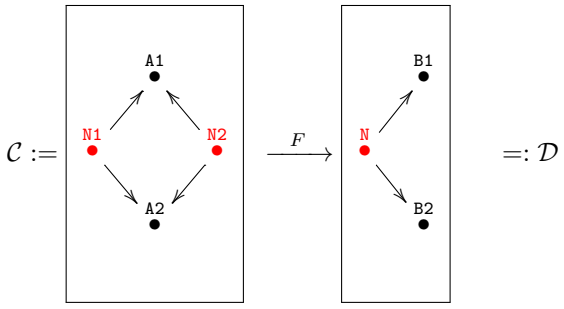
Typed instances form a category, and two instances are *equivalent*, written $\cong$, when they are isomorphic objects in this category. Isomorphism of typed instances captures our expected notion of equality on typed instances, where the "structure parts" are compared for isomorphism and the "attribute parts" are compared for equality under such an isomorphism.

4.3 Functorial Data Migration

In this section we define the original signature morphisms and data migration functors of Spivak's [74], as well as "typed signature morphisms" and "typed data migration functors", which are our extension of Spivak's [74] to attributes. The basic idea is that associated with a sufficiently well-behaved mapping between signatures $F : S \to T$ is a data transformation $\Delta_F : T\text{-}\mathbf{Inst} \to S\text{-}\mathbf{Inst}$ and left and right adjoints $\Sigma_F, \Pi_F : S\text{-}\mathbf{Inst} \to T\text{-}\mathbf{Inst}$. See chapter 1 for a formal definition of adjoint.

4.3.1 Signature Morphisms

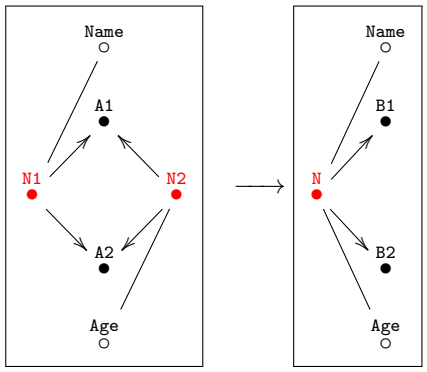
Let $\mathcal{C}$ and $\mathcal{D}$ be signatures. A *signature morphism* $F\colon \mathcal{C} \to \mathcal{D}$ is a mapping that takes vertices in $\mathcal{C}$ to vertices in $\mathcal{D}$ and arrows in $\mathcal{C}$ to *paths* in $\mathcal{D}$; in so doing, it must respect arrow sources, arrow targets, and path equivalences. In other words, if $p_1 \cong p_2$ is a path equivalence in $\mathcal{C}$, then $F(p_1) \cong F(p_2)$ is a path equivalence in $\mathcal{D}$. Each signature morphism $F\colon \mathcal{C} \to \mathcal{D}$ determines a unique *schema morphism* $[\![F]\!]\colon [\![\mathcal{C}]\!] \to [\![\mathcal{D}]\!]$ in the obvious way. Two signature morphisms $F_1\colon \mathcal{C} \to \mathcal{D}$ and $F_2\colon \mathcal{C} \to \mathcal{D}$ are *equivalent*, written $F_1 \cong F_2$, if they denote isomorphic functors. Below is an example of a signature morphism.



In the above example, the nodes `N1` and `N2` are mapped to `N`, the two morphisms to `A1` are mapped to the morphism to `B1`, and the two morphisms to `A2` are mapped to the morphism to `B2`.

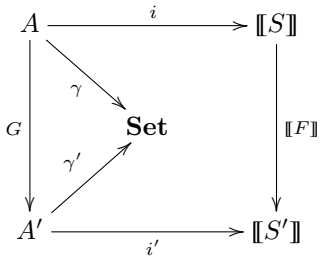
4.3.2 Typed Signature Morphisms

Intuitively, signature morphisms are extended to typed signatures by providing an additional mapping between attributes. For example, we might have `Name` and `Age` attributes in our source and target typings:



In the above, we map `Name` to `Name` and `Age` to `Age`. More complicated mappings of attributes are also possible, as we will see in the next section.

Formally, let S be a signature and $[\![\Gamma]\!] = (A, i, \gamma)$ a typing for S. Let S' be a signature and $[\![\Gamma']\!] = (A', i', \gamma')$ a typing for S'. A typed signature morphism from (S,Γ) to (S',Γ') consists of a signature morphism $F : S \to S'$ and a functor $G : A \to A'$ such that the following diagram commutes:



4.3.3 Data Migration Functors

Each signature morphism $F\colon \mathcal{C} \to \mathcal{D}$ is associated with three adjoint data migration functors, Δ_F, Σ_F, and Π_F, which migrate instance data on $\mathcal{D}$ to instance data on $\mathcal{C}$ and vice versa. A summary is given here:

Data migration functors induced by a translation $F : \mathcal{C} \to \mathcal{D}$				
Name	**Symbol**	**Type**	**Idea of definition**	**Relational**
Pullback	Δ_F	$\Delta_F \colon \mathcal{D}$–**Inst** $\to \mathcal{C}$–**Inst**	Composition with F	Project
Right Pushforward	Π_F	$\Pi_F \colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst**	Right adjoint to Δ_F	Join
Left Pushforward	Σ_F	$\Sigma_F \colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst**	Left adjoint to Δ_F	Union

Definition 1 (Data Migration Functors)**.** *Let $F \colon \mathcal{C} \to \mathcal{D}$ be a functor. We will define a functor $\Delta_F \colon \mathcal{D}$–**Inst** $\to \mathcal{C}$–**Inst**; that is, given a $\mathcal{D}$-instance $I \colon \mathcal{D} \to$ **Set** we will construct a $\mathcal{C}$-instance $\Delta_F(I)$. This is obtained simply by composing the functors I and F to get*

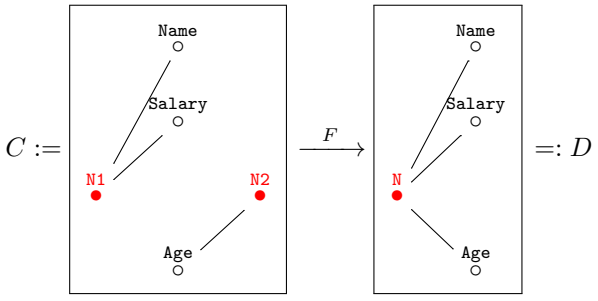
$$\Delta_F(I) := I \circ F \colon \mathcal{C} \to \textbf{Set} \qquad \mathcal{C} \xrightarrow{F} \mathcal{D} \xrightarrow{I} \textbf{Set}, \quad \mathcal{C} \xrightarrow{\Delta_F I} \textbf{Set}$$

*Then, $\Sigma_F \colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst** is defined as the* left adjoint *to Δ_F, and $\Pi_F \colon \mathcal{C}$–**Inst** $\to \mathcal{D}$–**Inst** is defined as the* right adjoint *to Δ_F.*

Data migration functors extend to *typed data migration functors* over typed instances in a natural way. We will use typed signatures and instances as we examine each data migration functor in turn below.

4.3.4 Δ

Consider the following signature morphism $F : C \to D$:



Even though our translation F points forwards (from C to D), our migration functor Δ_F points "backwards" (from D-instances to C-instances). Consider the instance J, on schema D, defined by

$J :=$

N			
ID	**Name**	**Age**	**Salary**
1	Bob	20	$250
2	Sue	20	$300
3	Alice	30	$100

$\Delta_F(J)$ splits up the columns of table N according to the translation F, resulting in

$I :=$

N1		
ID	**Name**	**Salary**
1	Bob	$250
2	Sue	$300
3	Alice	$100

N2	
ID	**Age**
a	20
b	20
c	30

Because of the globally unique ID requirement, the IDs of the two tables N1 and N2 must be disjoint. Note that Δ never changes the number of IDs associated with a node. For example, table N1 is ostensibly the "projection" of Age, yet has two rows with age 20.

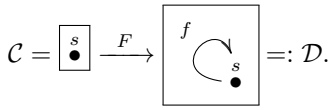
4.3.5 Π

Consider the morphism $F : C \to D$ and C-instance I defined in the previous section. Our migration functor Π_F points in the same direction as F: it takes C-instances to D-instances. In general, Π will take the join of tables. We can Π along any typed signature morphism whose attribute mapping is a bijection. Continuing with our example, we find that $\Pi_F(I)$ will take the cartesian product of N1 and N2:

N			
ID	**Name**	**Age**	**Salary**
1	Alice	20	$100
2	Alice	20	$100
3	Alice	30	$100
4	Bob	20	$250
5	Bob	20	$250
6	Bob	30	$250
7	Sue	20	$300
8	Sue	20	$300
9	Sue	30	$300

This example illustrates that adjoints are not, in general, inverses. Intuitively, the above instance is a product rather than a join because in there is no path between `N1` and `N2`.

Remark. When the target schema is infinite, on finite inputs Π may create uncountably infinite result instances. Consider the unique signature morphism



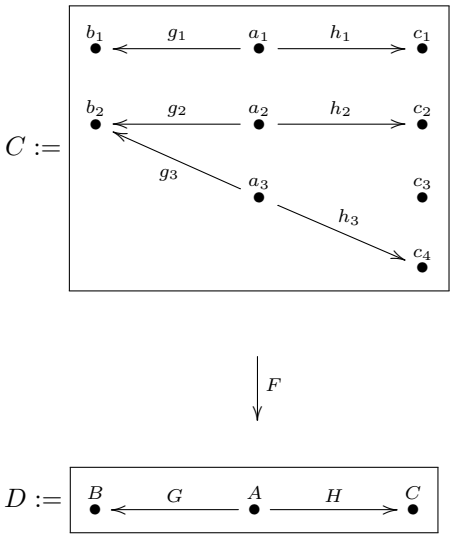
Here $[\![\mathcal{D}]\!]$ has arrows $\{f^n \mid n \in \mathbb{N}\}$ so it is infinite. Given the two-element instance $I : \mathcal{C} \to \mathbf{Set}$ with $I(s) = \{\text{Alice}, \text{Bob}\}$, the rowset in the right pushforward $\Pi_F(I)$ is the (uncountable) set of infinite streams in $\{\text{Alice}, \text{Bob}\}$, i.e. $\Pi_F(I)(s) = I(s)^{\mathbb{N}}$.

4.3.6 Σ

For the purposes of this chapter we will define Σ_F only for functors F that are *discrete op-fibrations*. Inasmuch as Σ can be thought of as computing unions, functors that are discrete op-fibrations intuitively express the idea that all such unions are over "union compatible" tables.

Definition 2 (Discrete op-fibration)**.** *A functor $F : \mathcal{C} \to \mathcal{D}$ is called a* discrete op-fibration *if, for every object $c \in \text{Ob}(\mathcal{C})$ and every arrow $g : d \to d'$ in $\mathcal{D}$ with $F(c) = d$, there exists a unique arrow $\bar{g} : c \to c'$ in $\mathcal{C}$ such that $F(\bar{g}) = g$.*

Consider the following discrete op-fibration, which maps as to A, bs to B, cs to C, gs to G, and hs to H:



Intuitively, $F\colon \mathcal{C} \to \mathcal{D}$ is a discrete op-fibration if "the columns in each table T of D are exactly matched by the columns in each table in C mapping to T." Since $a_1, a_2, a_3 \mapsto A$, they must have the same column structure and they do: each has two non-ID columns. Similarly each of the b_i and each of the c_i have no non-ID columns, just like their images in D.

To explain the action of Σ_F, consider the following instance:

a1		
ID	**g1**	**h1**
11	7	1

a2		
ID	**g2**	**h2**
16	9	3
15	10	4
14	8	4

a3		
ID	**g3**	**h3**
13	10	17
12	9	18

b1
ID
7
6

b2
ID
10
9
8

c1
ID
2
1

c2
ID
4
3

c3
ID
5

c4
ID
18
17

The result of Σ_F is:

A		
ID	**G**	**H**
16	9	3
15	10	4
14	8	4
13	10	17
12	9	18
11	7	1

B
ID
10
9
8
7
6

C
ID
18
17
5
4
3
2
1

By counting the number of rows it is easy to see that Σ computes union: `A` has $6 = 1 + 3 + 2$ rows, `B` has $5 = 3 + 2$ rows, `C` has $7 = 2 + 2 + 1 + 2$ rows.

We can Σ along any typed signature morphism for which the attribute mapping is also "union compatible in the sense of Codd": string attributes must map to string attributes, etc. Intuitively, the attribute data will be unioned together in exactly the same way as ID data.

Technically, Σ is a *disjoint* union. However, by requiring our IDs to be globally unique, we can use regular union to implement disjoint union: the globally unique ID assumption ensures that for all distinct tables X, Y in a functorial instance, $|X \cup Y| = |X| + |Y|$

Remark. In this chapter we require that signature morphisms used with Σ be discrete op-fibrations. However, it is possible to define Σ for arbitrary, un-restricted signature morphisms, at the following cost:

- Unrestricted Σs may not exist for typed instances.
- An unrestricted variant of FQL will probably not be closed under composition.
- To implement unrestricted Σ we may be required to synthesize new IDs, and termination of this process is semi-decidable.

4.4 FQL

The goal of this section is to define and study an algebraic query language where every query denotes a composition of data migration functors. Our syntax for queries is designed to build-in the syntactic restrictions discussion in the previous section and to provide a convenient normal form.

Definition 3 (FQL Query)**.** *A* FQL query Q *from* S *to* T*, denoted* $Q\colon S \rightsquigarrow T$ *is a triple of typed signature morphisms* (F, G, H)*:*

$$S \xleftarrow{F} S' \xrightarrow{G} S'' \xrightarrow{H} T$$

such that

- $[\![S]\!], [\![S']\!], [\![S'']\!]$*, and* $[\![T]\!]$ *are finite*
- G*'s attribute mapping is a bijection*
- H *is a discrete op-fibration with a union compatible attribute mapping*

Semantically, the query $Q\colon S \rightsquigarrow T$ corresponds to a functor $[\![Q]\!]\colon [\![S]\!]$–**Inst** $\to [\![T]\!]$–**Inst** given as follows:

$$[\![Q]\!] := \Sigma_{[\![H]\!]}\Pi_{[\![G]\!]}\Delta_{[\![F]\!]}\colon [\![S]\!]\text{–}\mathbf{Inst} \to [\![T]\!]\text{–}\mathbf{Inst}$$

By choosing two of F, G, and H to be the identity mapping, we can recover Δ, Σ, and Π. However, grouping Δ, Σ, and Π together like this formalizes a query as a disjoint union of a join of a projection. (Interestingly, in the SPCU relational algebra, the order of join and projection are swapped: the normal form is that of unions of projections of joins.)

Theorem 1 (Closure under composition)**.** *Given any FQL queries $f\colon S \rightsquigarrow X$ and $g\colon X \rightsquigarrow T$, we can compute an FQL query $g \cdot f\colon S \rightsquigarrow T$ such that*

$$[\![g \cdot f]\!] \cong [\![g]\!] \circ [\![f]\!]$$

Example

We now present an example FQL program using FQL's concrete syntax (this example is built in to the FQL IDE as the "FOIL" example). The basic idea is that starting from a schema with four nodes, a, b, c, d, we can compute an output instance with $(|a| + |b|) \times (|c| + |d|)$ rows by first unioning (with Σ) and then taking products (with Π). By the closure under composition theorem, we know that we may instead first take products and then union, and compute the same instance with an equivalent $|a| \times |c| + |a| \times |d| + |b| \times |c| + |b| \times |d|$ rows.

```
schema Begin = {
 nodes
   a,b,c,d;
 attributes; arrows; equations;
}

schema Added = {
 nodes
  aPLUSb,cPLUSd;
 attributes; arrows; equations;
}

schema Multiplied = {
 nodes
  aPLUSbTIMEScPLUSd;
 attributes; arrows; equations;
}
```

```
mapping F = {
 nodes
  a -> aPLUSb,
  b -> aPLUSb,
  c -> cPLUSd,
  d -> cPLUSd;
 attributes;
 arrows;
} : Begin -> Added

mapping G = {
 nodes
  aPLUSb -> aPLUSbTIMEScPLUSd,
  cPLUSd -> aPLUSbTIMEScPLUSd;
 attributes;
 arrows;
} : Added -> Multiplied

// Below, put any number of elements into a,b,c,d.
// The output should have (a+b)*(c+d) many elements

instance I = {
 nodes
  a -> {1},
  b -> {1,2},
  c -> {1,2},
  d -> {1,2,3};
 attributes; arrows;
}: Begin

instance J = sigma F I
```

```
instance K = pi G J


mapping idB = id Begin
mapping idA = id Added
mapping idM = id Multiplied
query p = delta idB pi idB sigma F
query q = delta idA pi G sigma idM
query res = (p then q)

instance resinst = eval res I
```

4.5 SQL Generation

In this section we define SQL generation algorithms for Δ, Σ, and Π. Let $F\colon S \to T$ be a typed signature morphism.

4.5.1 Δ

Theorem 2. *We can compute a SQL program $[F]_\Delta\colon T \to S$ such that for every T-instance $I \in T$–***Inst***, we have $\Delta_F(I) \cong [F]_\Delta(I)$.*

We sketch the algorithm as follows. We are given a T-instance I, presented as a set of binary functions, and are tasked with creating the S-instance $\Delta_F(I)$. We describe the result of $\Delta_F(I)$ on each table in the result instance by examining the schema S:

- for each node N in S, the binary table $\Delta_F(N)$ is the binary table $I_{F(N)}$
- for each attribute A in S, the binary table $\Delta_F(A)$ is the binary table $I_{F(A)}$
- Each edge $E : X \to Y$ in S maps to a path $F(E) : FX \to FY$ in T. We compose the binary edges tables making up the path $F(E)$, and that becomes the binary table $\Delta_F(E)$.

The SQL generation algorithm for Δ sketched above does not maintain the globally unique ID requirement. For example, Δ can copy tables. Hence we must also generate SQL to restore this invariant.

4.5.2 Σ

Theorem 3. *Suppose F is a discrete op-fibration and has a union compatible attribute mapping. Then we can compute a SQL program $[F]_\Sigma \colon S \to T$ such that for every S-instance $I \in S$–**Inst**, we have $\Sigma_F(I) \cong [F]_\Sigma(I)$.*

We sketch the algorithm as follows. We are given a S-instance I, presented as a set of binary functions, and are tasked with creating the T-instance $\Sigma_F(I)$. We describe the result of $\Sigma_F(I)$ on each table in the result instance by examining the schema T:

- for each node N in T, the binary table $\Delta_F(N)$ is the union of the binary node tables in I that map to N via F.
- for each attribute A in T, the binary table $\Delta_F(A)$ is the union of the binary attribute tables in I that map to A via F.
- Let $E : X \to Y$ be an edge in T. We know that for each $c \in F^{-1}(X)$ there is at least one path p_c in C such that $F(p_c) \cong e$. Compose p_c to a single binary table, and define $\Sigma_F(E)$ to be the union over all such c. The choice of p_c will not matter.

4.5.3 Π

Theorem 4. *Suppose $[\![S]\!]$ and $[\![T]\!]$ are finite, and F has a surjective attribute mapping. Then we can compute a SQL program $[F]_\Pi \colon S \to T$ such that for every S-instance $I \in S$–**Inst**, we have $\Pi_F(I) \cong [F]_\Pi(I)$.*

The algorithm for Π_F is more complicated than for Δ_F and Σ_F. In particular, its construction makes use of comma categories, which we have not yet defined, as well as "limit tables", which are a sort of "join all". We define these now.

Let B be a typed signature and H a typed B-instance. The limit table lim_B is computed as follows. First, take the cartesian product of every binary reflexive node table in B, and naturally join the attribute tables of B. Then, for each edge $e : n_1 \to n_2$ filter the table by $n_1 = n_2$. This filtered table is the limit table lim_B.

Let $S : A \to C$ and $T : B \to C$ be functors. The comma category $(S \downarrow T)$ has for objects triples (α, β, f), with α an object in A, β and object in B, and $f : S(\alpha) \to T(\beta)$ a morphism in C. The morphisms from (α, β, f) to (α', β', f') are all the pairs (g, h) where $g : \alpha \to \alpha'$ and $h : B \to B'$ are morphisms in A and B respectively such that $T(h) \circ f = f' \circ S(g)$.

The algorithm for Π_F proceeds as follows. First, for every object $d \in T$ we consider the comma category $B_d := (d \downarrow F)$ and its projection functor $q_d : (d \downarrow F) \to C$. (Here we treat d as a functor from the empty category). Let $H_d := I \circ q_d : B_d \to \mathbf{Set}$, constructed by generating SQL for $\Delta(I)$. We say that the limit table for d is $lim_{B_d} H_d$, as described above. Now we can describe the target tables in T:

- for each node N in T, generate globally unique IDs for each row in the limit table for N. These GUIDs are $\Pi_F(N)$.
- for each attribute $A : X \to type$ in T, $\Pi_F(A)$ will be a projection from the limit table for X.
- for each edge $E : X \to Y$ in T, $\Pi_F(E)$ will be a map from X to Y obtained by joining the limit tables for X and Y on columns which "factor through" E.

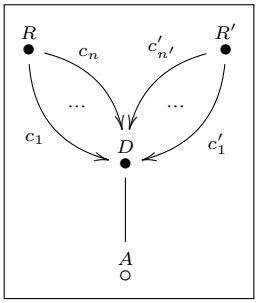
Remark 1. Our SQL generation algorithms for Δ and Σ work even when $[\![S]\!]$ and $[\![T]\!]$ are infinite, but this is not the case for Π. To recover Π on infinite schemas, it is possible to target a category besides **Set**, provided that category is complete and co-complete. For example, it is possible to store our data not as relations, but as programs in the Turing-complete language PCF [74] [69].

Remark 2. Our SQL generation algorithms for Δ and Σ generate compositions along paths and unions thereof. As such, there is little room for optimization at the level of SQL generation. However, we have found that in practice, on even trivial examples, some basic SQL query planning optimizations, such as ordering joins based on the size of input relations, are required to get adequate performance. Indeed, running our generated SQL on a real-world SQL engine such as mySQL results in order of magnitude speed-ups compared to the naive SQL implementation built in to the FQL IDE. Our SQL generation strategy for Π does (sometimes) result in redundant computation, and can be optimized along the lines proposed for right Kan-extensions in [35].

4.6 SQL in FQL

Because FQL operates on functorial instances, it is not possible to implement many relational operations directly in FQL. However, we can always "encode" arbitrary relational databases as functorial instances.

Relational signatures are encoded as "pointed" signatures with a single attribute that can intuitively be thought of as the active domain. For example, the signature for a relational schema with two relations $R(c_1, \ldots, c_n)$ and $R'(c'_1, \ldots, c'_{n'})$ has the following form:

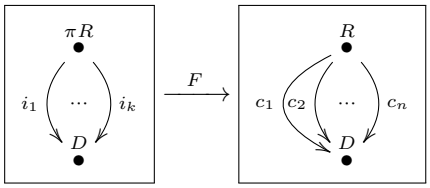


We might expect that the $c_1, \ldots, c_n$ would be attributes of node R, and hence there would be no node D, but that doesn't work: attributes may not be joined on. Instead, we must think of each column of R as a mapping from R's domain to IDs in D, and A as a mapping from IDs in D to constants. We will write $[R]$ for the encoding of a relational schema R and $[I]$ for the encoding of a relational R-instance I.

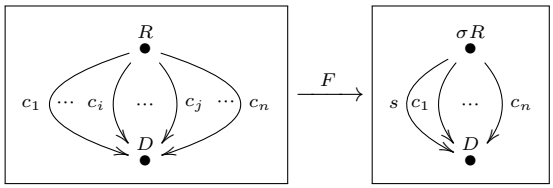
4.6.1 Conjunctive queries (Bags)

FQL can implement relational conjunctive (SPC/select-from-where) queries *under bag semantics* directly using the above encoding. In what follows we will omit the attribute A from the diagrams. For simplicity, we will assume the minimal number of tables required to illustrate the construction. We may express the (bag) operations $\pi, \sigma, \times$ as follows:

- Let R be a table. We can express $\pi_{i_1,\ldots,i_k} R$ using the pullback Δ_F, where F is the following functor

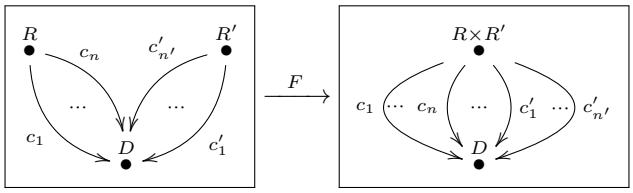


- Let R be a table. We can express $\sigma_{i=j} R$ using $\Delta_F \Pi_F$, where F is the following functor:



  Here $F(c_i) = F(c_j) = s$. If we wanted the more economical query in which the column s is not duplicated, we would use Π_F instead of $\Delta_F \Pi_F$.

- Let R and R' be tables. We can express $R \times R'$ as the right pushforward Π_F, where F is the following functor:



Theorem 5 (Conjunctive RA in FQL (bags))**.** *Let R be a relational schema, and I an R-instance. For every conjunctive (SPC) query q under bag semantics we can compute a FQL program $[q]$ such that $[q(I)] \cong [q]([I])$.*

4.6.2 Conjunctive Queries (Sets)

The encoding strategy described above fails for relational conjunctive queries under their set-theoretic semantics. For example, consider a simple two column relational table R, its encoded FQL instance $[R]$, and an attempt to project off col1 using Δ:

R	
col1	**col2**
x	y
x	z

$[R]$		
ID	**col1**	**col2**
0	x	y
1	x	z

$\Delta[R]$	
ID	**col1**
0	x
1	x

This answer is incorrect under projection's set-theoretic semantics because it has the wrong number of rows. However, it is possible to extend FQL with an operation, $relationalize_T : T\text{-}\mathbf{Inst} \to T\text{-}\mathbf{Inst}$, such that FQL+*relationalize* can implement every relational conjunctive query under normal set-theoretic semantics. Intuitively, $relationalize_T$ converts a T-instance to a smaller T-instance by equating IDs that cannot be distinguished. The relationalization of the above instance would be

$relationalize(\Delta[R])$	
ID	**col1**
0	x

which is correct for the set-theoretic semantics. We have the the following theorem:

Theorem 6 (Conjunctive RA in FQL (sets))**.** *Let R be a relational schema, and I an R-instance. For every conjunctive (SPC) query q under set semantics we can compute a FQL program $[q]$ such that $[q(I)] \cong relationalize_T([q]([I]))$.*

Provided T is obtained from a relational schema (i.e., has the pointed form described in this section), $relationalize_T$ can easily be implemented using the same SPCU+GUIDgen operations as Δ, Σ, Π.

4.6.3 Union

The bag union of two relational instances R_1 and R_2 can be expressed using disjoint union as $R_1 + R_2$; similarly, the set union can be expressed as $relationalize(R_1 + R_2)$. Hence, given the results of the previous sections, to translate the SPCU relational algebra to FQL it is sufficient to implement disjoint union using FQL. However, because FQL queries are unary and union is binary, implementing disjoint union using FQL requires encoding two C-instances as an instance on the co-product schema $C + C$.

For any two signatures S, T, the co-product signature $S + T$ is formed by taking the disjoint union of S, T's nodes, attributes, edges, and equations. An S instance can be injected into $(S + T)$–**Inst** using $\Sigma_F : S$–**Inst** $\to (S + T)$–**Inst**, where $F : S \to S + T$ is the canonical inclusion map. We have the following theorem:

Theorem 7 (Co-products in FQL)**.** *Let C be a signature and $I : C$–**Inst** and $J : C$–**Inst** be instances. Then the co-product instance $I + J : C$–**Inst** is expressible as $\Sigma_F(K) : C$–**Inst**, where $F : C + C \to C$ is the "fold" signature morphism taking each copy of C in the co-product schema $C + C$ to C, and $K : (C + C)$–**Inst** is formed by injecting $I : C$–**Inst** and $J : C$–**Inst** into $(C + C)$–**Inst**.*

4.7 Schema Transformation

In this section we prove that the functorial data model is a *schema transformation framework* in the sense of Alagic and Bernstein [3] [10]. In fact, their notion is essentially equivalent to that of *institution* [40].

It is easy to describe basic functorial data migration using Alagic and Bernstein's categorical model theory [3]. However, there is a fundamental difference in terminology. In this chapter, we use "signature" for a finitely presented category, and "schema" for the potentially infinite category it denotes. Alagic and Bernstein leave "signature" undefined, and use "schema" for a pair of a signature and a set of sentences in some logical formalism. Hence, every schema in our sense can be represented as a potentially infinite schema in their sense (its presentation), and every signature in our sense can be represented as a finite schema in their sense (its finite presentation). We consider path-equivalences to be part of our signatures, so our instances obey path-equivalences by definition; in their system, path-equivalences are not part of signatures and a separate satisfaction relation $\models_S$ characterizes the path-equivalences that hold in an S-instance. In the following definition, we use "signature" and "schema" in their sense. Let L be a logical formalism such as first-order logic. Then

Definition 4. *A schema transformation framework is a tuple* $(\mathbf{Sig}, Sig_0, Sen, Db, \models)$ *such that*

1. **Sig** *is the category of signatures together with their morphisms, and* **Sig** *has an initial object* Sig_0*.*
2. $Sen : \mathbf{Sig} \to \mathbf{Set}$ *is a functor such that* $Sen(Sig)$ *is the set of all L-sentences over the signature* Sig*.*
3. Db: $\mathbf{Sig} \to \mathbf{Cat}^{op}$ *is a functor sending each signature* Sig *to the category* $Db(Sig)$ *of* Sig*-instances and their morphisms.*
4. *For each signature* Sig*, the satisfaction relation* $\models_{Sig} \subset |Db(Sig)| \times Sen(Sig)$ *is such that for each schema signature morphism* ϕ: $Sig_A \to Sig_B$*, the following integrity requirement holds for each* Sig_B *database* d_B *and each sentence* $e \in Sen(Sig_A)$:

$$d_b \models_{Sig_B} Sen(\phi)(e) \quad \textit{iff} \quad Db(\phi)(d_b) \models_{Sig_A} e$$

Theorem 8. *The functorial data model is a schema transformation framework.*

We use the following definitions:

1. **Sig** is a the category of finitely presented freely generated categories, and Sig_0 is the empty category.
2. $Sen(Sig)$ is the set of all equations over the signature Sig.
3. Db: $\mathbf{Sig} \to \mathbf{Cat}^{op}$ is Δ.
4. $I \models p_1 \cong p_2$ is defined in the intuitive way.

4.8 FQL and EDs

In this section we discuss with some in-progress work about implementing FQL using embedded dependencies (EDs) [2]. We begin by noting some technical complications that arise when relating FQL to EDs. The first is that untyped instances in FQL correspond to relational instances that are made up entirely of meaningless globally unique IDs. If we think of FQL IDs as labelled nulls [27], then every natural transformation is a homomorphism, and vice versa. However, the correct notion of equivalence to use for untyped FQL instances is isomorphism, not homomorphic equivalence (instances I and J are homomorphically equivalent when there are homomorphisms $h : I \to J$ and $h' : J \to I$, but h and h' need not be inverses).

The second complication is that the constraints required to hold of functorial instances, e.g., (4.1)(4.2), are not all EDs. In particular, the "globally unique ID assumption" must be stated using negation:

$$\forall x, N_1(x,x) \to \neg N_2(x,x) \wedge \neg N_3(x,x) \wedge \dots$$

However, the unique ID assumption is an artifact of our SQL generation strategy, rather than a requirement of functorial data migration itself. In particular, the unique ID assumptions lets us use relational union to implement disjoint union.

Bearing these two complications in mind, we conjecture the following: every un-restricted Σ can be implemented as the *initial solution* to a set of EDs; every Π can be implemented as the *terminal solution* to a set of EDs; and every Δ can be implemented as the initial *and* terminal solution to a set of EDs. We now explain this terminology. If φ is a set of EDs and I an FQL instance, a *solution* to (φ, I) is an instance U such that $U \models \phi$ and there exists a natural transformation $h : I \to U$. U is *initial* if, for every other solution U' such that $h' : I \to U'$, there exists a unique natural transformation $f : U \to U'$ such that $h' = h; f$. Dually, U is *terminal* if it has a unique commuting natural transformation $h : U' \to U$ for every other solution U'.

We implement Δ, Σ, Π with EDs as follows. Let $F : C \to D$ be a signature morphism. We define the disjoint union signature $C + D$ by taking the disjoint union of C and D's nodes, attributes, arrows, and equations. Then we define the signatures $C \star_\Sigma D, C \star_\Pi D, C \star_\Delta D$ by adding additional paths and equations to $C + D$:

- the arrows of $C \star_\Pi D$ contain, for each each node $c \in C$, an arrow $m_c : F(c) \to c$; the equations of $C \star_\Pi D$ contain, for every arrow $f : c \to c'$ in C, the equation $F(f); m_{c'} = m_c; f$, and for every attribute a from c in C, the equation $F(a) = m_c; a$.
- the arrows of $C \star_\Sigma D$ contain, for each each node $c \in C$, an arrow $l_c : c \to F(c)$; the equations of $C \star_\Sigma D$ contain, for every arrow $f : c \to c'$ in C, the equation $l_c; F(f) = f; l_{c'}$, and for every attribute a from c in C, the equation $a = l_c; F(a)$.
- $C \star_\Delta D$ contains the l_c and m_c described above, the equations described above, and the additional equations $m_c; l_c = id$ and $l_c; m_c = id$.

The set of EDs that implements Δ, Σ, Π is then simply the functorality EDs from (4.1)(4.2) and the path equality constraints of $C \star D$, which are easy to express as EDs. Operationally, we start with a C (resp, D)

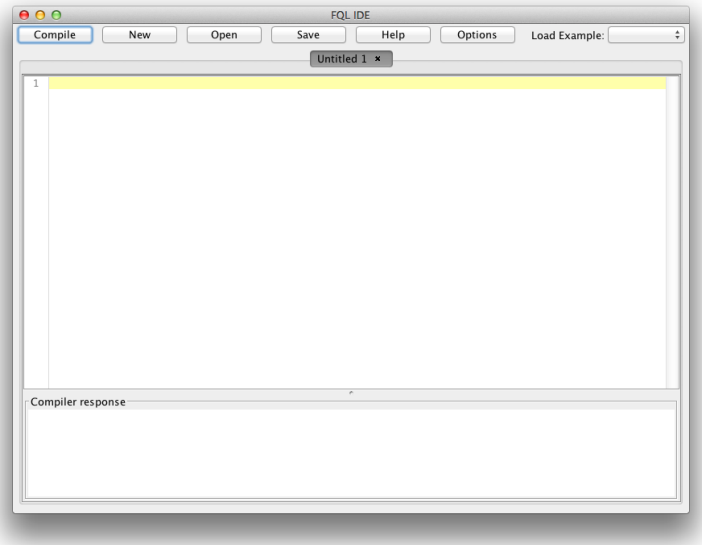
instance I, we compute an initial or terminal $C \star D$ solution IJ, and the desired D (resp, C) instance J will be a subset of the tables of IJ.

We have modified the FQL compiler to emit the EDs described above, and to solve them using the standard chase and the core chase [31]. We find that, on every example we have tried, the standard chase correctly computes Σ and Δ data migrations. The core chase does not, because the core of a chased solution I will only be homomorphically equivalent to I, not isomorphic to I, as we require.

Running the chase on the EDs generated for Π always results in an empty target instance, because the "existential force" of the EDs for Π is target-to-source. A category-theoretic understanding is that for our purposes, the standard chase computs initial instances, whereas our Π migrations are terminal instances. We are unaware of any traditional relational algorithm that implements terminal solutions to EDs. However, because we can implement Π using SPC+keygen, it is likely that Π can be implemented with a different set of EDs than those described in this section.

4.9 FQL Tutorial

In this section we present a tutorial about FQL, available at wisnesky.net/fql.html. The FQL compiler emits SQL (technically, PSM) code that can be run on any RDBMS. The FQL compiler is hosted inside of an integrated development environment, the FQL IDE. The FQL IDE is an open-source Java program that provides a code editor for and visual representation of FQL programs. A screen shot of the initial screen of the FQL IDE is shown below.



The FQL IDE is a multi-tabbed text file editor that supports the usual operations of saving, opening, copy-paste, etc. Associated with each editor is a "compiler response" text area that displays the SQL output of the FQL compiler (invoked with the "compile" button), or, if compilation fails, an error message. The built-in FQL examples can be loaded by selecting them from the "load example" combo box in the upper-right. In the rest of this tutorial we will refer to these examples.

4.9.1 FQL Syntax

An FQL program is an ordered list of named *declarations*. Each declaration defines either a *schema*, an *instance*, a *mapping*, or a *query*. We now describe each of these concepts in turn. Comments in FQL are Java style, either "//" or "/* */".

Schemas

Select the "employees" examples. This will create a new tab containing the following FQL code:

```
schema S = { nodes Employee, Department;

 attributes
  name  : Department -> string,
  first : Employee -> string,
  last  : Employee -> string;


 arrows
  manager   : Employee -> Employee,
  worksIn   : Employee -> Department,
  secretary : Department -> Employee;


 equations
  Employee.manager.worksIn = Employee.worksIn, //1
  Department.secretary.worksIn = Department, //2
  Employee.manager.manager = Employee.manager; //3
}
```

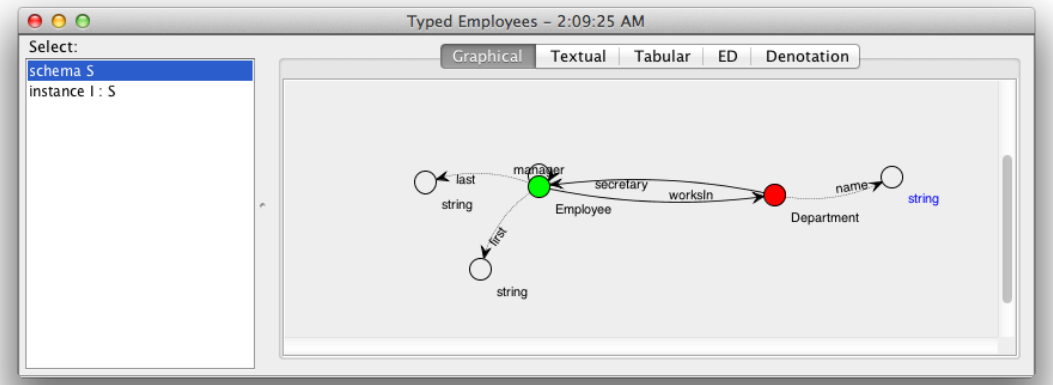
This declaration defines a schema S consisting of two *nodes*, three *attributes*, three *arrows*, and three *equations*. In relational terminology, this means that

- Each node corresponds to an *entity type*. In this example, the entities are employees and departments.
- A node/entity type may have any number of attributes. Attributes correspond to observable atoms of type int or string. In this example, each department has one attribute, its name, and each employee has two attributes, his or her first and last name.
- Each arrow $f : X \to Y$ corresponds to a function f from entities of type X to entities of type Y. In this example, manager maps employees to employees, worksIn maps employees to departments, and secretary maps departments to employees.
- The equations specify the data integrity constraints that must hold of all instances that conform to this schema. FQL uses equalities of paths as constraints. A path p is defined inductively as

$$p ::= node \mid p.arrow$$

  Intuitively, the meaning of "." is composition. In this example, the constraints are: 1) every employee must work in the same department as his or her manager; 2) every departmental secretary must work for that department; and 3) there are employees and managers, but not managers of managers, managers of managers of managers, etc.

The FQL IDE can render schemas into a graphical form similar to that of an entity-relationship (ER) diagram. Press "compile", and select the schema S from the viewer:



Note that the four sections, “nodes”, “attributes”, “arrows”, and “equations” are ended with semi-colons, and must appear in that order, even when a section is empty. The “denotation” tab prints the category that the schema denotes.

Instances

Continuing with the built-in “employees” example, we see that it also contains FQL code that defines an instance of the schema S defined in the previous section:

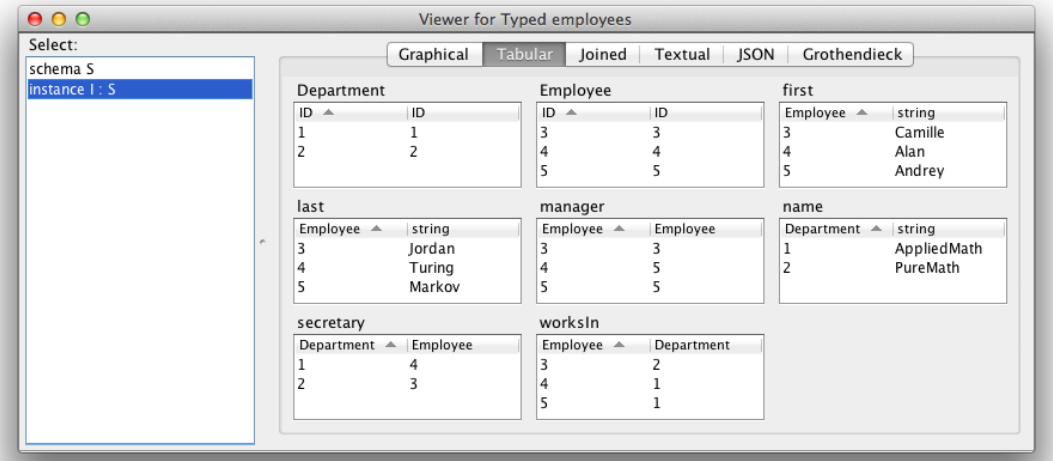
```
instance I : S = {
 nodes
  Employee -> { 101, 102, 103 },
  Department -> { q10, x02 };

 attributes
  first -> { (101, Alan), (102, Camille), (103, Andrey) },
  last  -> { (101, Turing), (102, Jordan), (103, Markov) },
  name  -> { (q10, AppliedMath), (x02, PureMath) };

 arrows
  manager -> { (101, 103), (102, 102), (103, 103) },
  worksIn -> { (101, q10), (102, x02), (103, q10) },
  secretary -> { (q10, 101), (x02, 102) };
}
```

This declaration defines an instance I that conforms to schema S. This means that

- To each node/entity type corresponds a set of globally unique IDs. In this example, the employee IDs are 101, 102, and 103, and the departmental IDs are q10 and x02.
- Each attribute corresponds to a function that maps IDs to atoms. In this example, we see that employee 101 is Alan Turing, employee 102 is Camille Jordan, employee 103 is Andrey Markov, department q10 is AppliedMath, and department x02 is PureMath.

- Each arrow $f : X \to Y$ corresponds to a function that maps IDs of entity type X to IDs of entity type Y. In this example, we see that Alan Turing and Andrey Markov work in the AppliedMath department, but Camille Jordan works in the PureMath department.

FQL assumes that every node, attribute, and arrow is stored as a binary table; node tables are stored as reflexive tables with types of the form (x, x). In addition, FQL assumes that the exact value of IDs are irrelevant, and in fact FQL will replace our IDs "q10, x02, 101, 102" and "103" with generated IDs 1,2,3,4,5. To visualize this instance, press "compile", select the instance from the viewer list, and click the "tabular" tab:



Mappings

Next, load the "delta" example. It defines two schemas, C and D, and a mapping F from C to D:

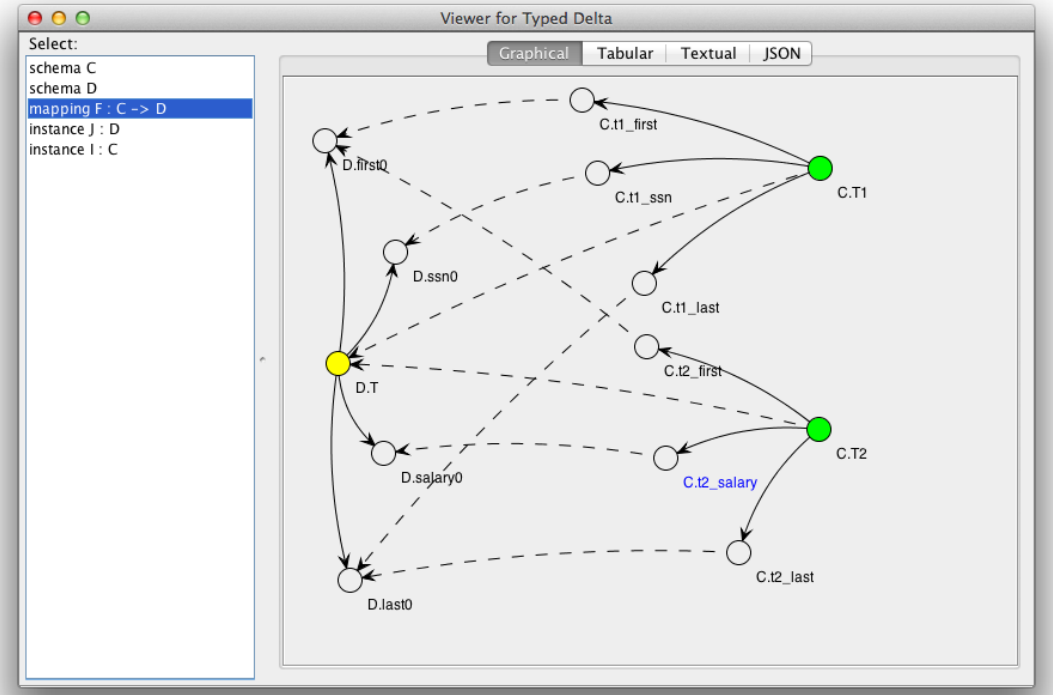
```
schema C = {
nodes T1, T2;
attributes
  t1_ssn    : T1 -> string,
  t1_first  : T1 -> string,
  t1_last   : T1 -> string,
  t2_first  : T2 -> string,
  t2_last   : T2 -> string,
```

```
   t2_salary : T2 -> int;
 arrows; equations;
}

schema D = {
 nodes T;
 attributes
  ssn0    : T -> string,
  first0  : T -> string,
  last0   : T -> string,
  salary0 : T -> int;
 arrows; equations;
}
mapping F : C -> D = {
 nodes T1 -> T, T2 -> T;
 attributes
  t1_ssn -> ssn0,  t1_first  -> first0,  t1_last -> last0,
  t2_last -> last0,  t2_salary -> salary0,  t2_first -> first0;
 arrows;
}
```

A mapping $F : C \to D$ consists of three parts:

- a mapping from the nodes in C to the nodes in D
- a mapping from the attributes in C to the attributes in D
- a mapping from the arrows in C to paths in D

A mapping must respect the equations of C and D: if p_1 and p_2 are equal paths in C, then $F(p_1)$ and $F(p_2)$ must be equal paths in D. If this condition is not met, FQL will throw an exception. Our example mapping is rendered in the viewer as follows:



An identity mapping can be formed using the keyword "id" as follows:

```
mapping F : C -> C = id C
```

Data Migration

Associated with a mapping $F : C \to D$ are three data migration operators:

- Δ_F, taking D instances to C instances, roughly corresponding to projection
- Π_F, taking C instances to D instances, roughly corresponding to join
- Σ_F, taking C instances to D instances, roughly corresponding to union

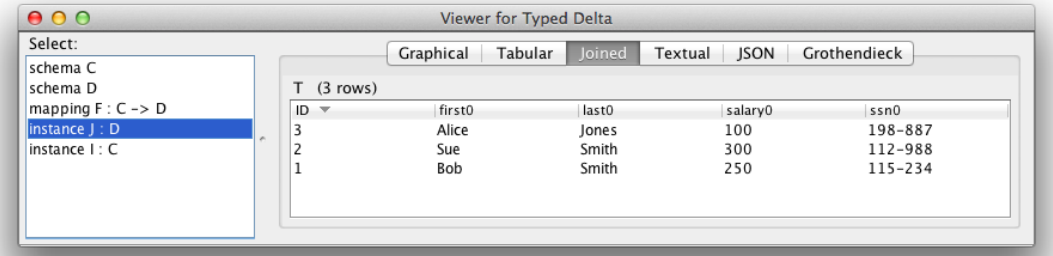
In general, certain restrictions must be placed on F to guarantee the above operations exist. We now describe each in turn.

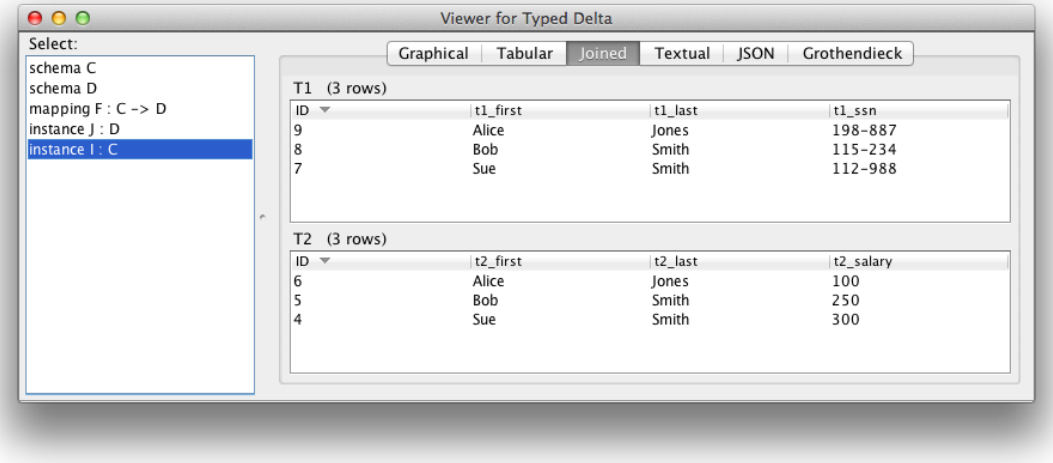
Delta

Continuing with the "delta" example, we see that the FQL program also defines a D-instance J, and computes $I := \Delta_F(J)$:

```
instance I : C = delta F J
```

Graphically, we have





In effect, we have projected the columns salary0 and last0 from J.

Pi

Load the "pi" example:

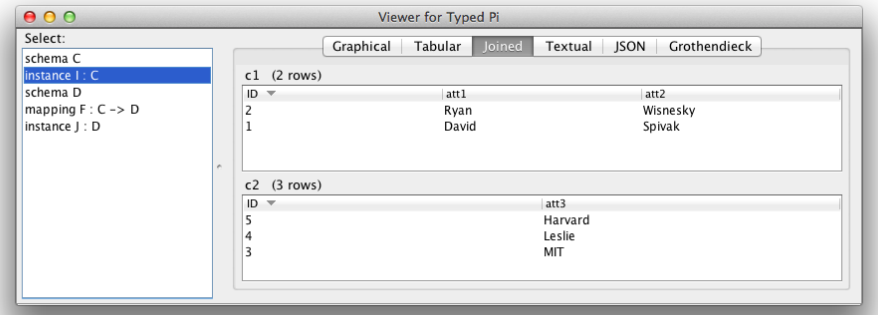
```
schema C = {
nodes c1,
 c2;
```

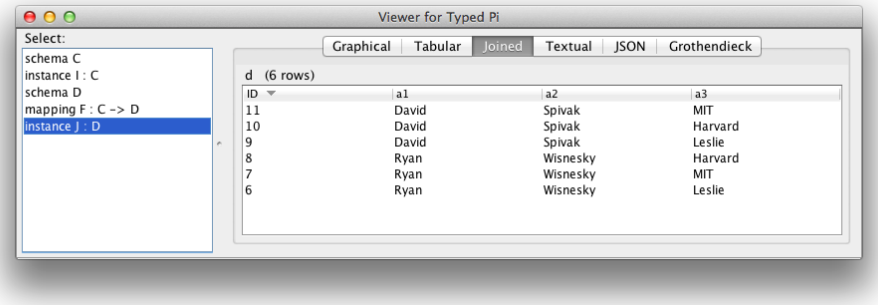
```
 attributes
att1 : c1 -> string,
att2 : c1 -> string,
att3 : c2->string;
 arrows;
 equations;
}


schema D = {
 nodes
  d;
 attributes
  a1 : d -> string,
  a2 : d -> string,
  a3 : d -> string;
 arrows;
 equations;
}

mapping F : C -> D = {
 nodes c1 -> d, c2 -> d;
 attributes att1 -> a1,  att2 -> a2, att3 -> a3;
 arrows;
}
```

This example defines an instance $I : C$ and computes $J := \Pi_F(I)$:

```
instance J : D = pi F I
```

Graphically, this is rendered as:





We see that we have computed the cartesian product of tables c_1 and c_2. Note that the attribute mapping part of F must be a bijection for Π_F to be defined; if this condition fails FQL will throw an exception.

Sigma

Load the "sigma" example:

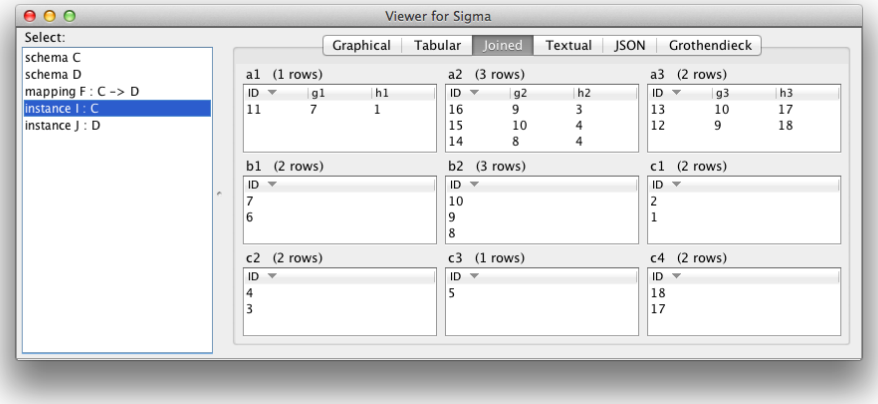
```
schema C = {
 nodes
  a1, a2, a3, b1, b2, c1, c2, c3, c4;

 attributes;

 arrows
  g1 : a1 -> b1,
  g2 : a2 -> b2,
  g3 : a3 -> b2,
```

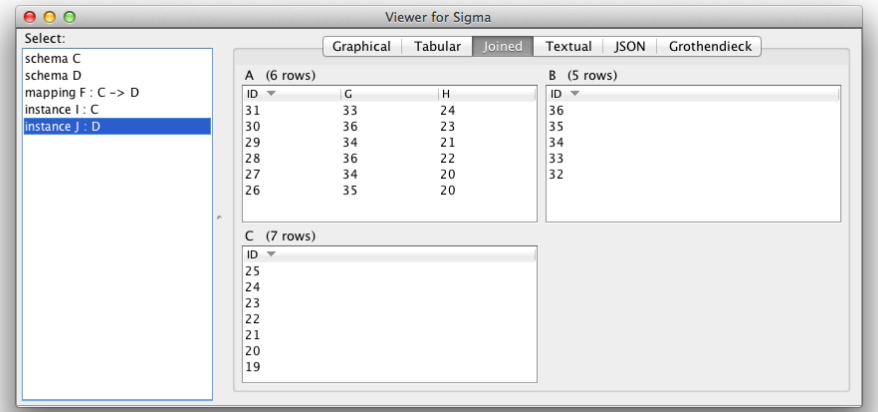
```
  h1 : a1 -> c1,
  h2 : a2 -> c2,
  h3 : a3 -> c4;

 equations;
}

schema D = {
 nodes A, B, C;

 attributes;

 arrows
  G : A -> B,
  H : A -> C;

 equations;
}

mapping F : C -> D = {
 nodes
  a1 -> A, a2 -> A, a3 -> A,
  b1 -> B, b2 -> B,
  c1 -> C, c2 -> C, c3 -> C, c4 -> C;

 attributes;

 arrows
  g1 -> A.G, g2 -> A.G, g3 -> A.G,
  h1 -> A.H, h2 -> A.H, h3 -> A.H;
}
```

This example defines an instance $I : C$ and computes $J := \Sigma_F(I)$:

```
instance J : D = sigma F I
```

Graphically, this is rendered as:





We see that we have computed union of tables a_1, a_2, and a_3 as A (6 rows), the union of tables b_1, b_2 as B (5 rows), and the union of c_1, c_2, c_3 and c_4 as C (7 rows).

To be defined, the functor F must satisfy the special condition of being a *discrete op-fibration*, which basically means "union compatible in the sense of Codd".

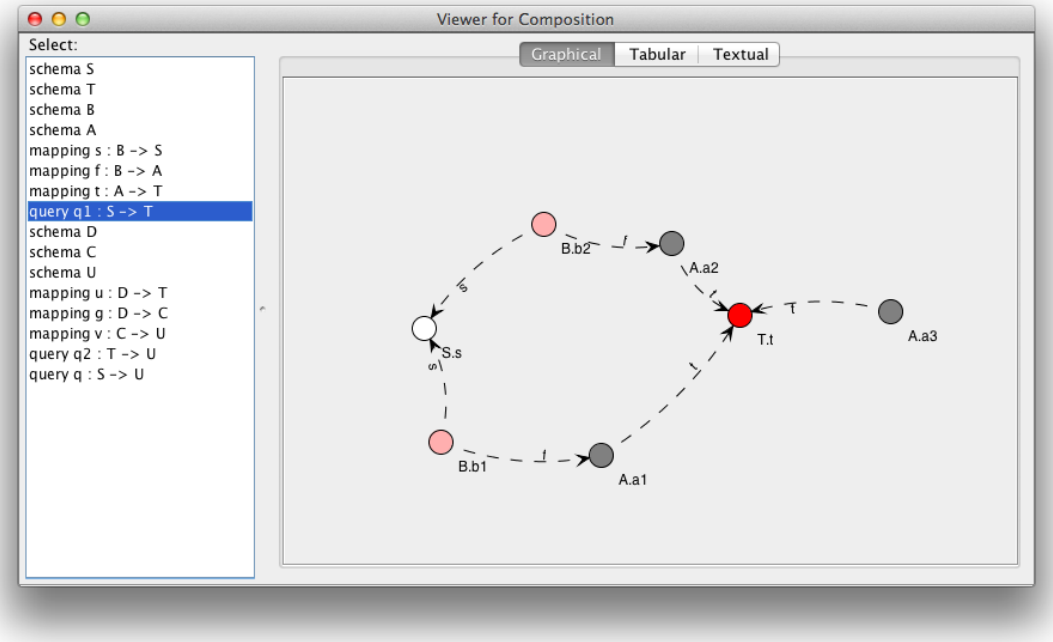
Queries

In SQL, unions of select-from-where clauses are the common programming idiom. In FQL, the common idiom is Σs of Πs of Δs. Load the "query composition" example:

```
schema S = { nodes s ; attributes; arrows; equations; }
```

```
schema T = { nodes t ; attributes; arrows; equations; }

schema B = { nodes b1,b2; attributes; arrows; equations; }

schema A = { nodes a1,a2,a3; attributes; arrows; equations; }


mapping s : B -> S = { nodes b1 -> s, b2 -> s; attributes; arrows; }

mapping f : B -> A = { nodes b1 -> a1, b2 -> a2 ; attributes; arrows; }

mapping t : A -> T = { nodes a1 -> t, a2 -> t, a3 -> t ; attributes; arrows; }


query q1 : S -> T = delta s pi f sigma t
```

In general, a query is simply a shorthand with special support for composition. Graphically, we have:



The four different colors in the viewer correspond to the four different schemas involved in a query. Queries may be evaluated using the keyword "eval":

```
instance J : S = ...

instance I : T = eval q1 J
```

Composition

FQL includes special support for composing queries. Continuing with the "query composition" example, we see that it defines another query:

```
schema D = { nodes d1,d2 ;  attributes; arrows; equations; }
schema C = { nodes c ;  attributes; arrows; equations; }
schema U = { nodes u ;  attributes; arrows; equations;}

mapping u : D -> T = { nodes d1 -> t, d2 -> t ;  attributes; arrows;}

mapping g : D -> C = { nodes d1 -> c, d2 -> c ;  attributes; arrows;}

mapping v : C -> U = { nodes c -> u ;  attributes; arrows; }

query q2 : T -> U = delta u pi g sigma v
```

We compose our two queries as follows:

```
query q : S -> U = q1 then q2
```

4.9.2 SQL Output

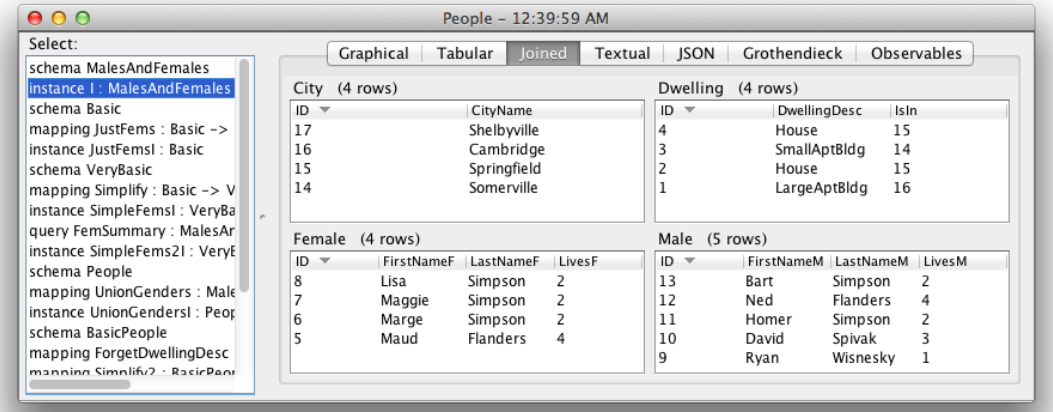
The FQL compiler emits (naive) SQL code that implements the FQL program. In fact, the FQL IDE executes the generated SQL to populate the viewer. The generated SQL may simply be copied into a command-line RDBMS top-level. For example, it can by executed by "mysql_embedded". However, to use the generated SQL correctly, note the following:

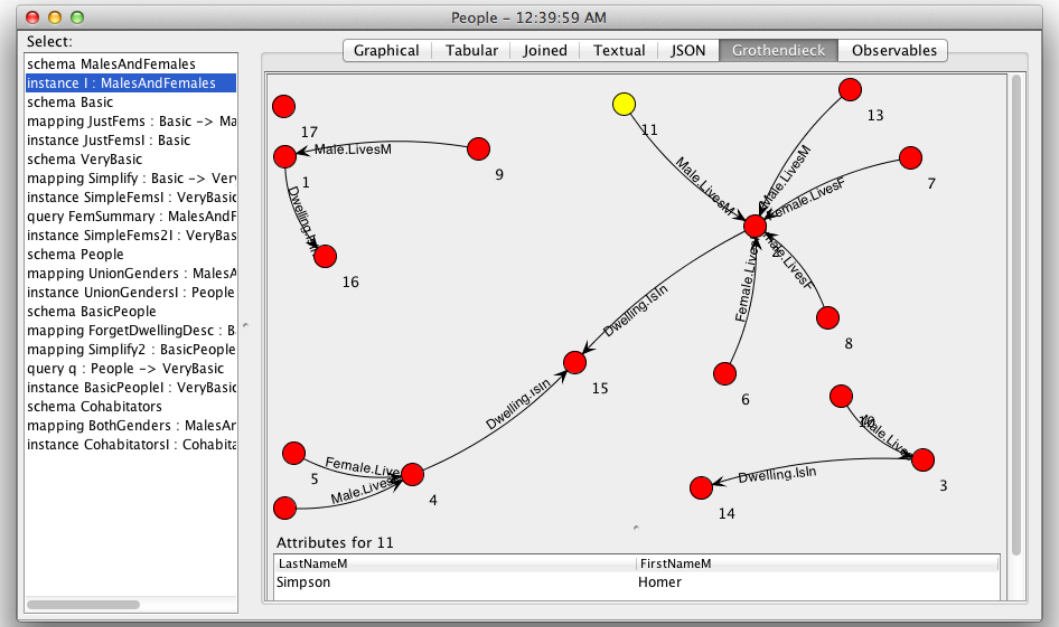
- A binary table R of an instance I is referred to as I_R. Hence, every node, attribute, and arrow in a schema must have a unique name. Names may appear in multiple schema, however.
- FQL is not case sensitive, but many SQL systems are. This may cause inadvertent name collisions.
- To use pre-existing database tables with the generated SQL output, CREATE TABLE commands in the generated SQL may need be suppressed using the "external" keyword. This mechanism is described in more detail in the "external" example.

4.9.3 Other Functionality

Category of Elements

The "Grothendieck" tab in an instance displays the instance as a category, "the category of its elements". In this view, nodes are entities and arrows are foreign-key correspondences. This is well illustrated using the "People" example:

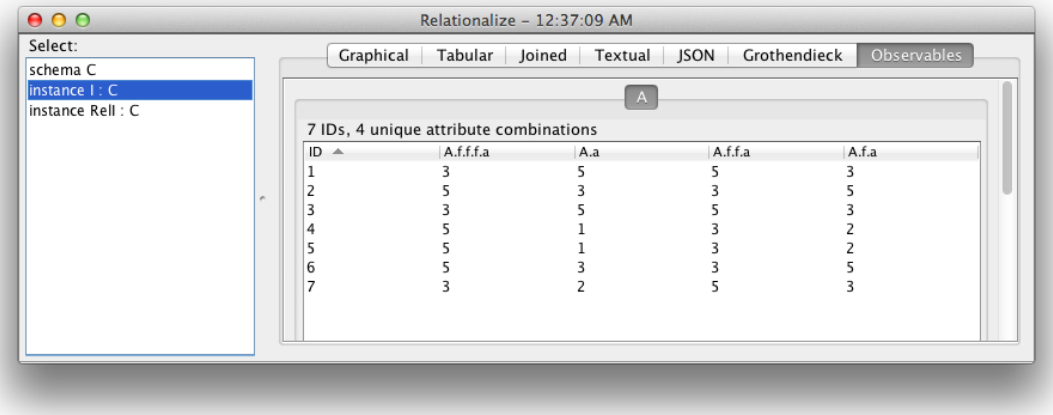




Relationalization and Observation

Associated with each type of entity in an instance is an "observation table". For an entity type/node N, the observation table joins together all attributes reachable by all paths out of N. Consider the "relationalize" example:

```
schema C={
 nodes A;
  attributes a:A->string;
  arrows f:A->A;
  equations A.f.f.f.f=A.f.f;
}

instance I:C={
nodes A->{1,2,3,4,5,6,7};
attributes a->{(1,1),(2,2),(3,3),(4,1),(5,5),(6,3),(7,5)};
arrows f->{(1,2),(2,3),(3,5),(4,2),(5,3),(6,7),(7,6)};
}

instance RelI:C=relationalize I
```



The operation "relationalize" will equate IDs that are not distinguished by attributes. In this example, 7 rows would collapse to 4. The relationalize operation is necessary to faithfully implement relational projection on relations that have been encoded as functorial instances.

4.10 FQL as a Functional Query Language

Up to this point in this chapter we have focused entirely on finitely presented schemas, mappings, and data migration functors. However, FQL is also a functional query language at two different levels: the level of schemas and mappings, and the level of instances and database morphisms. FQL possesses this structure because the category of categories is cartesian closed (and hence FQL schemas and mappings form a simply typed λ-calculus), and because for each signature T, the category of T-instances and their morphisms is a topos (and hence FQL T-instances and their database homomorphisms form a higher-order logic). For expediency, we ignore attributes in this section. The goal of this section is to define FQL as a type theory and formal language of categorical combinators [24].

Let $\mathcal{T}$ indicate finitely presented categories, $\mathcal{F}_{T_1,T_2} : T_1 \to T_2$ finitely presented functors, $\mathcal{I}_T$ finitely presented T-instances (functors from T to the category of sets), and $\mathcal{E}_{I_T^1,I_T^2} : I_T^1 \Rightarrow I_T^2$ finitely presented natural transformations (database homomorphisms from T-instances I_T^1 to I_T^2). The syntax of FQL types T, mappings F, T-instances I_T, and T-transformations (database homomorphisms over T-instances) E_T is given by the following grammar:

$$T ::= 0 \mid 1 \mid T + T \mid T \times T \mid T^T \mid \mathcal{T}$$

$$F ::= id_T \mid F; F \mid proj^1_{T,T} \mid proj^2_{T,T} \mid inj^1_{T,T} \mid inj^2_{T,T} \mid F \otimes F \mid F \oplus F \mid tt_T \mid ff_T \mid ev_{T,T} \mid \Lambda F \mid \mathcal{F}_{T,T}$$

$$I_T ::= 0_T \mid 1_T \mid I_T + I_T \mid I_T \times I_T \mid I_T^{I_T} \mid \mathcal{I}_T \mid \Omega_T \mid \Delta_F I \mid \Sigma_F I \mid \Pi_F$$

$$E_T ::= id_{I_T} \mid E_T; E_T \mid proj^1_{I_T,I_T} \mid proj^2_{I_T,I_T} \mid inj^1_{I_T,I_T} \mid inj^2_{I_T,I_T} \mid E_T \otimes E_T \mid E_T \oplus E_T$$

$$\mid ev_{I_T,I_T} \mid \Lambda E_T \mid \top_T \mid tt_{I_T} \mid ff_{I_T} \mid \mathcal{E}_{I_T^1,I_T^2} \mid eq_{I_T} \mid \Delta_F E \mid \Sigma_F E \mid \Pi_F E$$

4.10.1 Types

Types T are freely-generated by the grammar. Type isomorphism is given by:

$$T_1 \times (T_2 \times T_3) \cong (T_1 \times T_2) \times T_3 \qquad T_1 \times T_2 \cong T_2 \times T_1 \qquad T \times 1 \cong 1 \qquad 1^T \cong 1 \qquad T^1 \cong T$$

$$(T_1 \times T_2)^{T_3} \cong T_1^{T_3} \times T_2^{T_3} \qquad (T_1^{T_2})^{T_3} \cong T_1^{T_2 \times T_3} \qquad T_1 + (T_2 + T_3) \cong (T_1 + T_2) + T_3 \qquad T_1 + T_2 \cong T_2 + T_1$$

$$T \times 0 \cong 0 \qquad T + 0 \cong T \qquad T^0 \cong 1 \qquad T_1 \times (T_2 + T_3) \cong (T_1 \times T_2) + (T_1 \times T_3) \qquad T_1^{T_2+T_3} \cong T_1^{T_2} \times T_1^{T_3}$$

Isomorphism of objects in the free BCCC is not finitely axiomatizable, and its decidability is unknown [34]. For our purposes, isomorphism of finite categories is decidable.

4.10.2 Mappings

The typing rules for schema mappings F are exactly those of the internal language of a bi-cartesian closed category (i.e., $STLC^{0,1,+,\times}$), with constants $\mathcal{F}$:

$$\frac{}{id_T : T \to T} \qquad \frac{F : T_1 \to T_2 \qquad G : T_2 \to T_3}{F;G : T_1 \to T_3} \qquad \frac{}{tt_T : T \to 1} \qquad \frac{}{proj^1_{T_1,T_2} : T_1 \times T_2 \to T_1}$$

$$\frac{}{proj^2_{T_1,T_2} : T_1 \times T_2 \to T_2} \qquad \frac{F : T_1 \to T_2 \qquad G : T_1 \to T_3}{F \otimes G : T_1 \to T_2 \times T_3} \qquad \frac{}{ff_T : 0 \to T} \qquad \frac{}{inj^1_{T_1,T_2} : T_1 \to T_1 + T_2}$$

$$\frac{}{inj^2_{T_1,T_2} : T_2 \to T_1 + T_2} \qquad \frac{F : T_2 \to T_1 \qquad G : T_3 \to T_1}{F \oplus G : T_2 + T_3 \to T_1} \qquad \frac{}{ev_{T_1,T_2} : T_1^{T_2} \times T_2 \to T_1}$$

$$\frac{F : T_1 \times T_2 \to T_3}{\Lambda F : T_1 \to T_3^{T_2}} \qquad \frac{}{\mathcal{F}_{T_1,T_2} : T_1 \to T_2}$$

The equational theory for mappings F is exactly that of the internal language of a BCCC:

$$id; f = f \qquad f; id = f \qquad f; (g; h) = (f; g); h \qquad \Lambda ev = id \qquad \Lambda f \otimes a \; ; \; ev = id \otimes a; f \qquad f \otimes g; proj^1 = f$$

$$f \otimes g; proj^2 = g \qquad f; proj^1 \otimes f; proj^2 = f \qquad f = tt \quad (f : T \to 1) \qquad f = ff \quad (f : 0 \to T)$$

$$inj^1; f \oplus g = f \qquad inj^2; f \oplus g = g \qquad inj^1; f \oplus inj^2; f = f$$

4.10.3 Instances

For each T, T-instances I_T are freely generated by the grammar. More precisely, the typing rules for instances I_T are:

$$\frac{}{0_T : T - inst} \qquad \frac{}{1_T : T - inst} \qquad \frac{}{\mathcal{I}_T : T - inst} \qquad \frac{I_T : T - inst \qquad J_T : T - inst}{I_T + J_T : T - inst}$$

$$\frac{I_T : T - inst \qquad J_T : T - inst}{I_T \times J_T : T - inst} \qquad \frac{I_T : T - inst \qquad J_T : T - inst}{I_T^{J_T} : T - inst} \qquad \frac{F : T_1 \to T_2 \qquad I : T_2 - inst}{\Delta_F I : T_1 - inst}$$

$$\frac{F : T_1 \to T_2 \qquad I : T_1 - inst}{\Sigma_F I : T_2 - inst} \qquad \frac{F : T_1 \to T_2 \qquad I : T_1 - inst}{\Pi_F I : T_2 - inst} \qquad \frac{}{\Omega_T : T - inst}$$

Instances obey the same equational theory as types, as well as additional equations we have omitted.

4.10.4 Transformations

For each T, the T-database homomorphisms (natural transformations) E_T are exactly those of the internal language of a topos (i.e., $STLC^{0,1,+,\times,\Omega}$ or higher-order logic from chapter 3). This language is essentially the same as F, extended with eq and $\top$. More precisely:

$$\frac{}{id_{I_T} : I_T \Rightarrow I_T} \qquad \frac{E : I_T^1 \Rightarrow I_T^2 \qquad E' : I_T^2 \Rightarrow I_T^3}{E; E' : I_T^1 \Rightarrow I_T^3} \qquad \frac{}{tt_{I_T} : I_T \Rightarrow 1_T} \qquad \frac{}{proj^1_{I_T^1, I_T^2} : I_T^1 \times I_T^2 \Rightarrow I_T^1}$$

$$\frac{}{proj^2_{I_T^1, I_T^2} : I_T^1 \times I_T^2 \Rightarrow I_T^2} \qquad \frac{E : I_T^1 \Rightarrow I_T^2 \qquad E' : I_T^1 \Rightarrow I_T^3}{E \otimes E' : I_T^1 \Rightarrow I_T^2 \times I_T^3} \qquad \frac{}{ff_{I_T} : 0_T \Rightarrow I_T} \qquad \frac{}{inj^1_{I_T^1, I_T^2} : I_T^1 \Rightarrow I_T^1 + I_T^2}$$

$$\frac{}{inj^2_{I_T^1, I_T^2} : I_T^2 \Rightarrow I_T^1 + I_T^2} \qquad \frac{E : I_T^2 \Rightarrow I_T^1 \qquad E' : I_T^3 \Rightarrow I_T^1}{E \oplus E' : I_T^2 + I_T^3 \Rightarrow I_T^1} \qquad \frac{}{ev_{I_T^1, I_T^2} : {I_T^1}^{I_T^2} \times I_T^2 \Rightarrow I_T^1}$$

$$\frac{E : I_T^1 \times I_T^2 \Rightarrow I_T^3}{\Lambda E : I_T^1 \Rightarrow {I_T^3}^{I_T^2}} \qquad \frac{}{\mathcal{E}_{I_T^1, I_T^2} : I_T^1 \Rightarrow I_T^2} \qquad \frac{}{eq_{I_T} : I_T \times I_T \Rightarrow \Omega_T} \qquad \frac{}{\top_T : 1_T \Rightarrow \Omega_T}$$

$$\frac{F : T_1 \to T_2 \qquad E : I^1_{T_2} \Rightarrow I^2_{T_2}}{\Delta_F E : \Delta_F I^1_{T_2} \Rightarrow \Delta_F I^2_{T_2}} \qquad \frac{F : T_1 \to T_2 \qquad E : I^1_{T_1} \Rightarrow I^2_{T_1}}{\Sigma_F E : \Sigma_F I^1_{T_1} \Rightarrow \Sigma_F I^2_{T_1}} \qquad \frac{F : T_1 \to T_2 \qquad E : I^1_{T_1} \Rightarrow I^2_{T_1}}{\Pi_F E : \Pi_F I^1_{T_1} \Rightarrow \Pi_F I^2_{T_1}}$$

Database homomorphisms obey the same equational theory as mappings, as well as additional equations we have omitted.

4.11 Conclusion

We are working to extend FQL with additional operations such as difference, selection by a constant, and aggregation. In addition, we are studying the systems aspects of FQL, such as the data structures and algorithms that would be appropriate for a native, non-SQL implementation of FQL. More speculatively, for every monad M in the category of sets, the functorial data model admits generalized M-instances [75], which are database instances where every foreign-key reference to a value of type t has been replaced by a value of type $M\ t$. We speculate that this additional structure can be used to extend FQL to handle purely functional implementations of monadic computational effects in the traditional style [65]. Finally, FQL queries where the source and target schemas have exactly one node can encode *polynomial functors* [36]. This opens the possibility of extending FQL with algebraic datatypes or recursive queries.

Chapter 5

Conclusion

We conclude with some thoughts on dependently-typed functional query languages (DT-FQLs). Given that dependent identity types can be used to represent data integrity constraints (chapter 2), it is likely that other dependent types can be used to represent other program properties useful for query processing. For example, we might allow users of a monad-based functional query language [17] to define their own monads and to prove that their definitions obey the monad laws. As another example, in languages that represent sets as lists, to correctly define a recursive function over a set we must ensure that the function does not depend on the order of the list elements [15]. Although checking such conditions is undecidable, proving such conditions is commonplace in dependently-typed languages such as Coq. We are thus hopeful that general dependent types can be used to good effect in functional query languages, but in obtaining the results described in this thesis we discovered two fundamental challenges that any DT-FQL must overcome.

First, users of a DT-FQL must be able to construct proof objects, either manually or automatically. Rather than build the significant amount of infrastructure required to program effectively with dependent types [11], a more lightweight approach would be to define a DT-FQL by giving a shallow embedding of the DT-FQL into a language like Coq, leaving "holes" for proof obligations to be discharged. Users of the DT-FQL could then program against a convenient surface syntax, but would also be able to rely on Coq's mature technology to ease the theorem proving burden. Unfortunately, reasoning about shallow embeddings can be considerably more difficult than reasoning about the embedded language directly [55]. We conclude that a better way to develop a DT-FQL would be as a domain-specific macro language inside of Coq [21]. Optimizations such as the chase (chapter 2) could be implemented as Coq plug-ins or tactics.

Second, building a traditional compiler for a DT-FQL requires a useful equational theory for the DT-FQL as well as a practical strategy for searching for equivalent programs. In practice, this requires being able to express the DT-FQL in an algebraic form. For example, the relational calculus compiles to the relational algebra, the nested relational calculus compiles to the nested relational algebra, and the simply-typed lambda calculus compiles to the categorical abstract machine language [24]. Unfortunately, algebraic formulations of many typed λ-calculi are currently unknown. We thus suggest that a DT-FQL adopt one of three possible compilation strategies: (1) compiling into the untyped SKI combinatory algebra [8], (2) compiling into an algebraic formulation of the calculus of constructions [73], or (3) compiling into the binary relation algebra [72] . In all cases it is unclear if the resulting equational theory will be useful in practice.

Bibliography

[1] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *The VLDB Journal*, 4(4):727–794, October 1995.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Suad Alagic and Philip A. Bernstein. A model theory for generic schema management. In *DBPL*, 2001.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification. Technical report, Sun Microsystems, Inc., 2007.

[5] Sergio Antoy and Michael Hanus. Functional logic programming. *Commun. ACM*, 53:74–85, April 2010.

[6] Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. Completion without failure, 1989.

[7] Sreeram Balakrishnan, Vivian Chu, Mauricio A. Hernández, Howard Ho, Rajasekar Krishnamurthy, Shi Xia Liu, Jan H. Pieper, Jeffrey S. Pierce, Lucian Popa, Christine M. Robson, Lei Shi, Ioana R. Stanoi, Edison L. Ting, Shivakumar Vaithyanathan, and Huahai Yang. Midas: integrating public financial data. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, 2010.

[8] H. P. Barendregt. *The lambda calculus : its syntax and semantics / H.P. Barendregt*. North-Holland Pub. Co. ; New York : New York, 1981.

[9] Michael Barr and Charles Wells, editors. *Category theory for computing science, 2nd ed.* 1995.

[10] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. SIGMOD '07, 2007.

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[12] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl C. Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[13] Richard Bird and Oege de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[14] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ICFP*, pages 213–225, 1996.

[15] Val Breazu-tannen, Peter Buneman, and Shamim Naqvi. Structural recursion as a query language. In *In Proceedings of 3rd International Workshop on Database Programming Languages*, pages 9–19, 1991.

[16] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *EDBT*, 1992.

[17] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1):87–96, 1994.

[18] S. Carmody, M. Leeming, and R.F.C. Walters. The todd-coxeter procedure and left kan extensions. *Journal of Symbolic Computation*, 19(5):459 – 488, 1995.

[19] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP*, 2007.

[20] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, 1977.

[21] Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 391–402. ACM, 2013.

[22] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, pages 65–98. Prentice-Hall, 1972.

[23] Ezra Cooper. The script-writer's dream: How to write great sql in your own language, and be sure it will succeed. In *DBPL*, pages 36–51, 2009.

[24] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173 – 202, 1987.

[25] Phillipe et al Cudre-Mauroux. A demonstration of scidb: A science-oriented dbms. In *VLDB: Proceedings of the VLDB Endowment*. VLDB Endowment, August 2009.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Comm.ACM*, 51(1):107–113, 2008.

[27] Alin Deutsch, Alan Nash, and Jeff Remmel. The chase revisited. PODS, 2008.

[28] Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Rec.*, 35:65–73, March 2006.

[29] Martin Erwig and Steve Kollmansberger. Probabilistic functional programming in haskell. *Journal of Functional Programming*, 2005.

[30] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[31] Ronald Fagin, Phokion G. Kolaitis, Alan Nash, and Lucian Popa. Towards a theory of schema-mapping optimization. PODS, 2008.

[32] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, December 2005.

[33] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. Reverse data exchange: Coping with nulls. In *Proceedings of the Twenty-eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '09, pages 23–32, 2009.

[34] Marcelo P. Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS, pages 147–, 2002.

[35] Michael Fleming, Ryan Gunther, and Robert Rosebrugh. A database of categories. *J. SYMBOLIC COMPUT*, 35:127–135, 2002.

[36] N. Gambino and J. Kock. Polynomial functors and polynomial monads.

[37] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.

[38] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of CS, University of Nottingham, November 1996.

[39] Neil Ghani and Patricia Johann. Monadic augment and generalised short cut fusion. *J. Funct. Program.*, 17(6):731–776, 2007.

[40] J.A. Goguen and R.M. Burstall. Introducing institutions. In *Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*. 1984.

[41] Torsten Grust. *Monad Comprehensions. A Versatile Representation for Queries. In The Functional Approach to Data Management, P.M.D. Gray and L. Kerschberg and P.J.H. King and A. Poulovassilis (eds.)*. Springer Verlag, 2003.

[42] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 1063–1066, 2009.

[43] Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.

[44] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, 2005.

[45] Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[46] Qi heng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of two semantic query optimization techniques in db2 universal database. VLDB, 1999.

[47] Fritz Henglein. Large-scale sound and precise program analysis: technical persepctive. *Commun. ACM*, 53:114–114, August 2010.

[48] Jieh Hsiang and Michael Rusinowitch. On word problems in equational theories. In *Automata, Languages and Programming*, volume 267 of *LNCS*. 1987.

[49] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[50] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09*, 2009.

[51] C. B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Euro-Par'97 Parallel Processing, volume 1300 of Lecture Notes in Computer Science*, pages 650–661. Springer, 1997.

[52] Michael Johnson and Robert Rosebrugh. Sketch data models, relational schema and data specifications. *Electronic Notes in Theoretical Computer Science*, 61(0):51 – 63, 2002.

[53] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. 1986.

[54] S. Kazem Lellahi and Val Tannen. A calculus for collections and aggregates. In *CTCS '97*, 1997.

[55] Daniel R. Licata and Robert Harper. A universe of binding and computation. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 123–134, 2009.

[56] Sam Lindley. Extensional rewriting with sums. In *In TLCA*, pages 255–271, 2007.

[57] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[58] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, 1991.

[59] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *FPCA: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 324–333, 1995.

[60] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 228–266, London, UK, 1995. Springer-Verlag.

[61] Sergey Melnik. *Generic Model Management: Concepts And Algorithms (Lecture Notes in Computer Science)*. 2004.

[62] Sergey Melnik, Philip A. Bernstein, Alon Halevy, and Erhard Rahm. Supporting executable mappings in model management. SIGMOD, 2005.

[63] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: a programming platform for generic model management. In *SIGMOD*, 2003.

[64] John C. Mitchell. *Foundations of programming languages*. MIT Press, Cambridge, MA, USA, 1996.

[65] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[66] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[67] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD 08*, 2008.

[68] Avi Pfeffer. Ibal: A probabilistic rational programming language. In *In Proc. 17th IJCAI*, pages 733–740, 2001.

[69] Gordon D. Plotkin. Lcf considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[70] Lucian Popa and Val Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *ICDT*, 1999.

[71] Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB, 2002.

[72] Vaughan Pratt. Origins of the calculus of binary relations. In *In Proceedings of MFCS'93*, pages 142–155, 1992.

[73] Eike Ritter. Categorical abstract machines for higher-order typed lambdacalculi. *Theoretical Computer Science*, 136(1):125 – 162, 1994.

[74] David I. Spivak. Functorial data migration. *Inf. Comput.*, 217:31–51, August 2012.

[75] David I. Spivak. Kleisli database instances, August 2012. http://arxiv.org/abs/1209.1011.

[76] David I. Spivak and Ryan Wisnesky. On the relational foundations of functorial data migration. http://arxiv.org/abs/1212.5303, 2012.

[77] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[78] Dan Suciu. Domain-independent queries on databases with external functions. In *In LNCS 893: Proceedings of 5th International Conference on Database Theory; 177–190*, pages 177–190. Springer, 1995.

[79] Val Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. ICDT '92, pages 140–154, London, UK, 1992. Springer-Verlag.

[80] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[81] Ryan Wisnesky. Minimizing monad comprehensions. Technical Report TR-02-11, Harvard University Computer Science Group. Available at ftp://ftp.deas.harvard.edu/techreports/tr-02-11.pdf, 2011.

[82] Limsoon Wong. *Querying nested collections*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1994. Supervisor-Buneman, Peter.

[83] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Prog*, 10, 1998.

[84] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.